
Let Run your Neurons
team
LRN

Graphics Programming with Direct X 9

Module II

(14 Week Lesson Plan)

Lesson 1: Meshes

Textbook: Chapter Eight (pgs. 2 – 77)

Goals:

The course begins by introducing some of the important mesh containers provided by the D3DX library. Discussion will center on performance issues, including attribute batching across mesh boundaries and subset rendering, as well as optimization techniques that speed up rendering on modern hardware. From there we will look at how to import X file geometry into our applications as well as how to construct and fill the mesh buffers manually. This will lead into a discussion of cloning (copying) mesh data and some of the features that can be exploited in the process. The next topic of discussion will be the management of geometric level of detail using view independent progressive meshes. We will look at how to construct and use progressive meshes and see how they work algorithmically. This will lead into an examination of one-off mesh simplification and how it can be done with D3DX support. We will conclude this lesson with a quick overview of a number of useful mesh utility functions.

Key Topics:

- ID3DXMesh Interface
  - Vertex/Index/Adjacency Buffers
  - Attribute Buffers and Subset Rendering
- Mesh Optimization
- ID3DXBuffer
- Mesh Loading
- Manual Mesh Creation
- Mesh Cloning
- ID3DXPMesh Interface
  - View Independent Progressive Meshes (VIPM)
  - Data Validation and Cleaning
  - Setting LOD
  - LOD Trimming
  - Vertex History
- ID3DXSPMesh Interface
- Global Mesh Utility Functions

Projects:

Lab Project 8.1: The CTriMesh Class (Mesh Viewer I)

Exams/Quizzes: NONE

Recommended Study Time (hours): 8 - 10

Lesson 2: Frame Hierarchies

Textbook: Chapter Nine (pgs. 2 – 87)

Goals:

In this lesson we will now look at how to import and manage more complex 3D models and scenes. We will introduce the concepts of frame of reference and parent-child hierarchical relationships and see how we can use these ideas to build elaborate scenes consisting of independent, animation-ready meshes. Early on in the process we will delve into the inner workings of X file templates to see how scene data is stored. This will set us up for a discussion of the very important D3DXLoadMeshHierarchyFromX function, which we will use many times in the coming lessons. Using this function properly will require an examination of the callback mechanisms and data structures used for application memory management. We will even talk about how to load custom data chunks. Finally, we will wrap up the lesson with a look at how to traverse, transform, and render a hierarchy of meshes. A very simple animation controller will be introduced during the process and in our lab project to setup our discussions in the next lesson.

Key Topics:

- Hierarchies
  - Frame of Reference
  - Parent/Child Relationships
- X File Templates
  - Open/Closed/Restricted Templates
  - Hierarchical X Files
- D3DXLoadMeshHierarchyFromX
- ID3DXAllocateHierarchy Interface
  - Allocating/De-allocating Frames
- ID3DXMeshContainer Interface
  - Allocating/De-allocating Mesh Containers
- Extending Hierarchy Data Types
- ID3DXLoadUserData Interface
  - Loading Custom Top-Level Data
  - Loading Customer Child Data
- Hierarchy Traversal and Rendering
- Simple Hierarchy Animation

Projects:

Lab Project 9.1: The CActor Class (Mesh Viewer II)

Exams/Quizzes: NONE

Recommended Study Time (hours): 10 - 12

Lesson 3: Keyframe Animation I

Textbook: Chapter Ten (pgs. 2 – 64)

Goals:

In this lesson our goal will be to learn the fundamentals of animating game scenes. The primary focus will be on using keyframe data to animate the hierarchies introduced in the previous lesson. Our initial discussions will take us back into the inner workings of X file templates, where we will learn about the various ways that animation data can be represented and how it all translates into D3DX data structures. From there we will begin our exploration of the powerful animation system available in DirectX. This exploration will involve understanding how the animation controller interpolates keyframe data and how that process can be controlled using various subsystems in the controller. Along the way we will examine the construction of a custom animation set object that can be plugged into the D3DX animation system.

Key Topics:

- Animation Blending
  - The Animation Mixer
    - Setting track weight, speed, priority
    - Enable/Disable Tracks
  - Priority Blending
- Animation Controller Cloning
- The Animation Sequencer
  - Registering Events
  - Event Handles
- The Animation Callback System
  - Callback keys and animation sets
  - Executing callback functions
  - ID3DXAnimationCallbackHandler Interface

Projects:

Lab Project 10.1: Animated CActor (Mesh Viewer III)
Lab Project 10.2: The Animation Splitter

Exams/Quizzes: NONE

Recommended Study Time (hours): 10 – 12

Lesson 4: Keyframe Animation II

Textbook: Chapter Ten (pgs. 64 – 114)

Goals:

In this lesson our goal will be to continue our discussion of animation fundamentals by examining some different animation controller subsystems. The first major controller subsystem encountered will be the animation mixer, where we will learn about the important topic of blending multiple simultaneous animations. After learning how to use the mixer and configure its tracks for blending, we will conclude our discussions by looking at how to setup various user-defined special events using both the animation sequencer and the animation callback system. These features will allow us to sync together our animation timeline with events like playing sound effects or triggering specific pieces of function code.

Key Topics:

- Animation Blending
  - The Animation Mixer
    - Setting track weight, speed, priority
    - Enable/Disable Tracks
  - Priority Blending
- Animation Controller Cloning
- The Animation Sequencer
  - Registering Events
  - Event Handles
- The Animation Callback System
  - Callback keys and animation sets
  - Executing callback functions
  - ID3DXAnimationCallbackHandler Interface

Projects:

Lab Project 10.1: Animated CActor (Mesh Viewer III) cont.
Lab Project 10.2: The Animation Splitter cont.

Exams/Quizzes: NONE

Recommended Study Time (hours): 10 – 12

Lesson 5: Skinning I

Textbook: Chapter Eleven (pgs. 2 – 115)

Goals:

In this lesson we will finally integrate animated game characters into our framework. This will build on all of the topics covered in the prior lessons including meshes, hierarchies, and the animation system. We will begin our examination by looking at some of the methods used for animating game characters in older games. What we learn will lead us straight into the idea of skinning and skeletal animation as a means for providing more realistic visual results. We will learn all about what skins and skeletons are, how they are constructed, and how they can be animated and rendered. As before we will look at the X file data templates and see how these translate into our game data structures. Then we will examine the various skinning options available via D3D. This will include detailed examinations of software skinning and hardware skinning; both non-indexed and palette-driven indexed skinning techniques.

Key Topics:

- Vertex Tweening
- Segmented Models and Animation
- Bone Hierarchies/Skeletons
- Vertex Blending
- Skinning
- X File Templates for Skinning
- The Bone Offset Matrix
- Software Skinning
- ID3DXSkinInfo Interface
- Non-Indexed Skinning
    - Setting multiple world matrices
    - Enabling/disabling vertex blending
    - ConvertToBlendedMesh
- Indexed Skinning
    - Determining Support
    - Matrix Palette Indices
    - ConvertToIndexedBlendedMesh
- Transforming and Rendering Skinned Characters

Projects:

Lab Project 11.1: Skinned CActor (Mesh Viewer IV)
Lab Project 11.2: The Animation Splitter II

Exams/Quizzes: NONE

Recommended Study Time (hours): 10 - 12

Lesson 6: Skinning II

Textbook: Chapter Twelve (pgs. 2 – 160)

Goals:

In this lesson we will conclude our exploration of skinning and animation by taking a different angle from the prior lesson. This time, rather than load skinned characters from an X file, we are going to construct an entire skeleton and skin model programmatically. The end result will be a robust tree class that we can use to generate realistic looking animated trees for populating our outdoor landscape scenes. Since this is the halfway point in the course, we are also going to make an effort to bring together much of what we have learned to date into a single demonstration lab project. One important focus in this second lab project will be the extension of our middle-tier to include data driven support between our application and the D3DX animation system. This upgraded system will handle animation switching, blending, and the other key areas that are necessary to simplify the communication pipeline between the application and the low level animation code. This will allow students to more easily integrate animation support into their game projects and have their AI or user-input systems interact and control the process.

Key Topics:

- Trees
  - Procedural Skins and Skeletons
  - Procedural Keyframe Animation
- The Animation Middle Layer
  - Data Driven File Support
  - Animation Set Blending
  - Controller Configuration
  - Playing Back Complex Animations

Projects:

Lab Project 12.1: The CTreeActor Class (Mesh Viewer V)
Lab Project 12.2: Summary Lab Project

Exams/Quizzes: NONE

Recommended Study Time (hours): 10 – 12

Lesson 7: Midterm Exam Preparation and Review

Textbook: Chapters 8 - 12

Goals:

The midterm examination in this course will consist of 50 multiple-choice and true/false questions pulled from the first five textbook chapters. Students are encouraged to use the lecture presentation slides as a means for reviewing the key material prior to the examination. The exam should take no more than two hours to complete. It is worth 30% of the final grade.

Office hours will be held for student questions and answers.

Key Topics:

Projects: NONE

Exams/Quizzes: Midterm Examination (50 questions)

Recommended Study Time (hours): 12 - 15

Lesson 8: Collision Systems I

Textbook: Chapter Thirteen

Goals:

In the second half of the course students will begin to explore important generic topics in the area of game engine design. While we will not conclude our game engine design studies until Module III, we will begin to lay the foundation for most of the core systems. In this lesson and the next we will undertake the development of a robust collision detection and response system. We begin with an overview of collision detection and look at the difference between broad and narrow phase algorithms. From there we will explore a sliding response system that is a staple of many first and third person games. After we have tackled the overall system architecture, including the management of geometry, we will introduce the concept of ellipsoid space and see how it will be used to facilitate the entire process. Then we will start our examination of the intersection algorithms that are going to be used in the narrow phase of our collision detection engine. We will talk about rays, what they are and how they can be tested against common game primitives. Then we will begin to look at how spheres can be tested against triangle interiors. This will lead into the additional testing algorithms covered in the next lesson.

Key Topics:

- Collision Systems Overview
- Broad Phase vs. Narrow Phase Collision Detection
- Collision Response
  - Sliding
- Ray Intersection Testing
  - Ray vs. Plane
  - Ray vs. Polygon
- Ellipsoids, Unit Spheres, and Ellipsoid Space
- Swept sphere vs. Triangle

Projects:

Lab Project 13.1: Collision System

Exams/Quizzes: NONE

Recommended Study Time (hours): 10 - 12

Lesson 9: Collision Systems II

Textbook: Chapter Thirteen

Goals:

In this lesson we will complete the development of our collision detection and response system. Since intersection testing is fundamentally about solving equations, we will begin by reviewing the concept of quadratic equations and some of the fundamental mathematics used during the detection phase. Quadratic equations are used in a number of our system's most important routines, so this overview should prove helpful to students who have forgotten some of this basic math. This will ultimately lead into an examination of intersection testing between swept spheres and the edges and vertices of our triangles. This is where we will wrap up our core routines for the narrow phase tests against static objects and environments. Once done, we will move on to discuss the means for colliding against dynamic objects that are part of the game environment. Dynamic object support will require a number of upgrades to both the detection and response stages in our code.

Key Topics:

- Quadratic Equations
- Swept Sphere Intersection Testing
  - Swept Sphere vs. Edge
  - Swept Sphere vs. Vertex
- Animation and the Collision Geometry Database
- Dynamic Object Collision Support

Projects:

Lab Project 13.1: Collision System

Exams/Quizzes: NONE

Recommended Study Time (hours): 10 - 12

Lesson 10: Spatial Partitioning I

Textbook: Chapter Fourteen

Goals:

In this lesson we will introduce some of the most important data structures and algorithms that are used to improve game engine performance. We will begin with simple axis-aligned hierarchical spatial partitioning data structures like quadtrees, octrees, and kD-trees. These systems will allow us to introduce broad phase collision detection into the system we developed in the prior lessons. Once done, we will introduce the very popular non axis-aligned spatial subdivision technique called binary space partitioning (BSP). BSP trees will actually be used in the next few lessons of the course to accomplish some very important ends, but for now only basic theory will be covered. The goal in our lab project will be to create a single base class with a core set of polygon querying functionality (intersection testing) and a set of derived classes for all tree types. This will allow students to mix and match the tree types as it suits the needs of their own projects.

Key Topics:

- Spatial Partitioning Data Structures/Algorithms
  - Quadtrees
  - Octrees
  - kD Trees
  - BSP Trees
- Polygon Clipping
- Polygon Database Intersection Querying
  - Ray/AABB/Sphere testing
- Broad Phase Implementation for Collision System

Projects:

Lab Project 14.1: Broad Phase Collision Detection

Exams/Quizzes: NONE

Recommended Study Time (hours): 10 - 12

Lesson 11: Spatial Partitioning II

Textbook: Chapter Fourteen

Goals:

In the last lesson we introduced some of the core partitioning structures that are used to improve game engine performance. Our prior focus was on the means for assembling the tree data structures and for traversing them to do polygon queries. This allowed us to add the very crucial broad phase to our collision detection system. In this lesson we will examine another aspect of using these trees – rendering. Hardware friendly rendering is an important idea that students need to think about when designing their partitioning systems. This can be a challenging thing to accomplish and there are many considerations, so this is something we will examine in this lesson. We will also introduce some additional concepts to speed up scene rendering by exploiting the hierarchical nature of our data during frustum culling and integrating the idea of frame coherence to add some additional optimization. Finally, we will look at using a BSP tree to perform accurate front to back sorting for rendering transparent polygons.

Key Topics:

- Accurate Alpha Polygon Sorting
- Frame Coherence
- Hardware Friendly Rendering
    - Static vs. Dynamic Solutions
    - Polygon Caches (Pros/Cons)
- Hierarchical Frustum Culling/Rendering

Projects:

Lab Project 14.2: Hierarchical Scene Rendering w/ BSP Alpha Sorting

Exams/Quizzes: NONE

Recommended Study Time (hours): 10 - 12

Lesson 12: Spatial Partitioning III

Textbook: Chapter Fifteen

Goals:

This lesson will conclude our studies of spatial partitioning techniques and begin the transition into the development of potential visibility sets. We will begin by learning how to add solid and empty space information to our BSP tree representation. This will provide the foundation for a number of important tasks that follow in this lesson and the next. With that code in place, we will look at how to build BSP trees out of scene brushes in order to perform constructive solid geometry (CSG). The CSG techniques we study will allow for merging together of geometric objects, carving shapes out of other shapes, and other interesting dynamic tasks. We will conclude the lesson by introducing the concept of portals, the first step towards development of PVS. We will look at how to generate them, split them, and remove any duplicates.

Key Topics:

- Solid Leaf BSP Trees
  - Rendering
  - Line of Sight
- Constructive Solid Geometry (CSG)
  - Union/Intersection/Difference Operations
- Portal Generation
  - Portal Splitting

Projects:

Lab Project 15.1: Solid Leaf Tree Compiler/Renderer
Lab Project 15.2: CSG Operations

Exams/Quizzes: NONE

Recommended Study Time (hours): 10 – 12

Lesson 13: Spatial Partitioning IV

Textbook: Chapter Fifteen

Goals:

Our final lesson will complete the development of our first performance oriented game engine design. We will begin by looking at the process of calculating potential visibility sets. The first step is the discussion of penumbras and anti-penumbras and how volumetric lighting models can be used to determine visibility. Using the portals created in the last lesson we will model the flow of light through the scene to mark off areas of shadow and light. This will provide enough information to be able to draw conclusions about visible areas in the level. After we have calculated our PVS, we will look at how to compress the information and then use it to efficiently render our scenes. We will conclude the lesson with a look at a different approach to BSP tree compilation. Since illegal geometry can corrupt the BSP building process, we will look at methods that allow us to avoid these problems and at the same time generate BSP trees and PVS for just about any type of scene our artists can create. These BSP trees will not use the level polygons as the means for compilation, but will instead use artist-generated simplified primitives that bound the actual level data. With fast rendering technology, efficient collision detection, spatial subdivision, and geometry and animation support all in place, students will be fully ready to wrap up their game engine development studies in Module III.

Key Topics:

- Potential Visibility Sets
  - Zero Run Length Encoding
  - Scene Rendering
- Anti-Penumbras
  - Generator Portal Visibility
  - Portal Flow
- Polygon-less BSP Trees
  - Illegal Geometry

Projects:

Lab Project 16.1: The BSP/PVS Compiler
Lab Project 16.2: Final Project

Exams/Quizzes: NONE

Recommended Study Time (hours): 12 - 15

Lesson 14: Final Exam Preparation and Review

Textbook: NONE

Goals:

The final examination in this course will consist of 75 multiple-choice and true/false questions pulled from all textbook chapters. Students are encouraged to use the lecture presentation slides as a means for reviewing the key material prior to the examination. The exam should take no more than three hours to complete. It is worth 70% of the final grade.

Office hours will be held for student questions and answers.

Key Topics:

Projects: NONE

Exams/Quizzes: NONE

Recommended Study Time (hours): 12 - 15

Chapter Eight

Meshes

Introduction

In Graphics Programming Module I we created our own mesh class (CMesh) to manage model geometry. While this is certainly an acceptable approach and can be useful for many tasks, the D3DX library provides a collection of mesh types that provide some key advantages over our simple vertex/index buffer wrapper class. This chapter will examine some of the core D3DX mesh types and we will learn how to work with them in our game development projects.

Some key features of D3DX provided meshes are:

Optimization: D3DX mesh objects are more than simple wrappers around a vertex and index buffer. They include important optimization features that allow for more efficient batching and rendering. These features alone make using D3DX mesh objects an attractive choice. Although D3DX mesh objects fully encapsulate batch rendering of their triangles, you are not required to use these features. If you wanted to store and render your meshes using proprietary mesh classes, you could use a D3DX mesh object to temporarily store and optimize your data for faster rendering with minimum state changes and better vertex cache coherency. Once done, you could lock the buffers and copy the optimized data back into your own mesh class and then destroy the D3DX mesh.

Asset Support: DirectX Graphics uses the X file as its native geometry file format for loading and storing 3D objects. Although we can store geometry in any file format we please, we are responsible for writing code to parse the data in that file and store it in our meshes. The X file format is flexible enough to store a wide variety of information, including complete scene hierarchies. While DirectX Graphics provides interfaces to help manually load and parse X file data structures, this is really not a task that will be relished by the uninitiated. Fortunately, we do not have to worry too much about that since the D3DX library provides functions that automate X file loading and data storage. With a single function call, we can load an X file and wind up with a 'ready to render' D3DX mesh, saving us a good deal of work. The X file format is now supported by most popular commercial 3D modeling applications, so we will be able to import high quality 3D artwork into our projects with ease. DirectX Graphics also ships (provided you also download the additional DirectX 9.0 Extras package) with command line tools that provide easy conversion of 3D Studio™ files (3ds) into the X file format. The *conv3ds.exe* is a command line tool that converts 3ds files to X files. Unfortunately these command line tools do not always work as well as one might hope. Instead of using the command line tools, DirectX Graphics now ships (provided once again that you download the Extras package) with various exporter plug-ins for popular graphics packages such as 3D Studio MAX™ and Maya™.

The Maya plug-in is called Xexport.mll. It is an mll (a Maya™ dynamic link library) that can be dragged and dropped into the plug-ins directory of the Maya™ application. This dll also ships with source code. The 3D Studio™ plug-in dll is called XskinExp.dle but using it is not quite as straightforward to integrate. First, only the source code is provided, so you will need to compile the dll yourself. Be sure that you have downloaded the 3D Studio MAX™ Software Development Kit and the Character Studio™ Software Development Kit for the source code to compile correctly (some versions of MAX may include the SDK on the CD). Unfortunately, one of the header files needed for the compile is missing from the DX9 extras package. To save confusion, we have supplied this missing file with the source code that accompanies this chapter.

> **Note**: GILES™ has the ability to import and export X files, so be sure to check out this feature if you choose to use this format in your applications. Since GILES™ imports .3ds files as well, it is worth considering as an alternative to the command line tools that ship with the DirectX SDK.

So even if you intend to use your own mesh classes for rendering, you can use a temporary D3DX mesh object to load in X file data. Again, it is little trouble to lock the mesh vertex and index buffers and copy out the data into your own mesh type. The result: fast and free X file loading for your application.

Utility: There are other useful features that become available when using D3DX meshes. For example there are intersection tests, bounding volume generation, vertex normal calculations, geometry cleaning functions to remove stray vertices, welding functions, level of detail algorithms, mesh cloning, and much more, all built right into the interfaces.

Although we are free to use the D3DX mesh solely as a utility object, often we will use all of its features: loading, optimization, and rendering. This will be the focus of our lab projects that accompany this chapter.

8.1 D3DX Mesh Types Overview

There are five mesh interfaces in D3DX, each providing a specific set of functionality. In this chapter we will discuss only four of these interfaces in detail: ID3DXBaseMesh, ID3DXMesh, ID3DXSPMesh and ID3DXPMesh. The remaining interface ID3DXPatchMesh, which manages curved surface rendering and related tasks, will be covered a bit later in this programming series.

Before getting under the hood with the four mesh types we are going to cover in this chapter, we will first briefly review the five D3DX mesh types and discuss the high level functionality they provide. After this brief overview, we will examine the mesh types in more detail and learn how to create them, optimize them, and render them.

8.1.1 ID3DXBaseMesh

ID3DXBaseMesh provides the inheritable interface for core mesh functionality. Features include vertex and index buffer management, mesh cloning, face adjacency information processing, rendering, and other housekeeping functions. Base meshes cannot be instantiated, so we will always create one of the derived mesh types -- the derived types support all of the base mesh interface methods.

> **Note:** While we will not use this interface directly, it serves a useful purpose in a game engine since we can store pointers of this type for all of our mesh objects. This allows us to query the interface to determine which of the other mesh types is actually being used. While we might use one or more derived class instances like ID3DXMesh and ID3DXPMesh in our scene, we can store pointers to each in an ID3DXBaseMesh array and cast between types as needed.

Because we will never explicitly instantiate an ID3DXBaseMesh, in this chapter we will cover its interface methods in the context of the more commonly used derived classes. We will indicate which functions belong to which interface as we progress, so that inherited functionality is not obscured by the examples. This will allow us to use code snippets that more closely reflect what we will see in our lab projects.

8.1.2 ID3DXMesh

This is the primary mesh container in DirectX and is the one we are likely to use most often. This mesh can be created and initialized manually (using functions like D3DXCreateMesh or D3DXCreateMeshFVF) or automatically as the result of file loading functions (like D3DXCreateMeshFromX). ID3DXMesh inherits all of the functionality of the ID3DXBaseMesh and provides additional functions for data optimization. Optimization involves algorithms for sorting vertex and index buffer data to provide maximum rendering performance.

8.1.3 ID3DXPMesh

ID3DXPMesh provides support for progressive meshes for run-time geometric level of detail (LOD) changes. Through this interface, the number of triangles used to render a model can be increased or decreased on the fly. The algorithm used is based on the VIPM (View Independent Progressive Mesh) technique. Models can have their triangles merged together to reduce the overall polygon count or alternatively, have their surface detail increased by using more of the original triangles in the rendered mesh. This allows us to adjust the detail level of the mesh to suit the needs of our application (ex. meshes further away from the camera might have polygonal detail reduced to speed up rendering).

8.1.4 ID3DXSPMesh

Simplification meshes provide the ability to reduce the vertex and/or triangle count in a model. Values can be provided by the application to specify which aspects of the model are more important than others during the reduction process. This provides a degree of control over which polygons are removed and which wind up being preserved in the final reduced model. Unlike progressive meshes, mesh simplification involves only face reduction and is a one-time-only operation. Since the results of the operation cannot be reversed, simplification is generally reserved for use in either a tool such as a model editor or as a one-time process that takes place during application initialization. This could be useful if you wanted to tailor your mesh triangle counts to suit the runtime environment. For example, you might wish to reduce the level of detail of certain meshes if you find that a software vertex processing device is all that is available on the current machine.

8.1.5 ID3DXPatchMesh

Patch meshes encapsulate the import, management, and manipulation of meshes which make use of higher-order curved surface primitives (called patches). A patch is defined by a series of control points describing the curvature of a surface. Because patch meshes store their data so differently from the other mesh representations, this interface does not inherit from ID3DXBaseMesh; it is derived directly from IUnknown. Most of the ID3DXBaseMesh interface methods would make little sense in terms of a patch mesh. Curved surfaces and higher-order primitives will be covered later in this series.

Let us now examine each mesh type in more detail.

8.2 ID3DXMesh

The ID3DXMesh interface is the basic mesh container in DirectX Graphics. As such, we begin our discussion by looking first at the internals of its data storage and move on to discuss its methods. There are four primary data storage buffers when working with meshes in DirectX: vertex buffers, index buffers, attribute buffers, and adjacency buffers. Let us look at each in turn.

8.2.1 The Vertex Buffer

D3DX meshes contain a single vertex buffer for storage of model vertex data. This is a standard IDirect3DVertexBuffer9 vertex buffer and is identical to the vertex buffers we have been using since Chapter Three. It can be created using any supported FVF, locked and unlocked, and read from and written to just like any other vertex buffer. All ID3DXBaseMesh derived interfaces inherit the LockVertexBuffer and UnlockVertexBuffer methods for obtaining direct access to the mesh's underlying vertex data.

```
HRESULT ID3DXMesh::LockVertexBuffer(DWORD Flags, VOID **ppData)
```

DWORD Flags
These are the standard locking flags that we have used when locking vertex buffers in previous lessons. The flags include `D3DLOCK_DISCARD`, `D3DLOCK_NOOVERWRITE`, `D3DLOCK_NOSYSLOCK`, `D3DLOCK_READONLY` and `D3DLOCK_NO_DIRTY_UPDATE` and can be used in combination to lock the buffer in an efficient manner (see Chapter Three).

VOID **ppData
This is the address of a pointer that will point to the vertex data if the lock is successful.

We release the lock on the mesh vertex buffer using the ID3DXMesh::UnlockVertexBuffer function.

```
HRESULT ID3DXMesh::UnlockVertexBuffer(VOID)
```

The following code snippet demonstrates mesh vertex buffer locking and unlocking. It is assumed that pd3dxMesh is a pointer to an already initialized ID3DXMesh.

```
ID3DXMesh *pd3dxMesh;

// Create the mesh here…

// Lock the vertex buffer and get a pointer to the vertex data
void *pVertices;
pd3dxMesh->LockVertexBuffer( 0, &pVertices );

// Fill the vertex buffer using the pointer

// When we are done, we must remember to unlock
pd3dxMesh->UnlockVertexBuffer();
```

8.2.2 The Index Buffer

D3DX meshes use a single index buffer to store model faces. In the context of the D3DX mesh functions, a face is always a triangle and the index buffer always contains indices that describe an indexed triangle list. Thus, if we called the ID3DXMesh::GetNumFaces method, we will be returned the total number of triangles in the mesh. This will always equal the total number of indices in the index buffer divided by three. D3DX meshes have no concept of quads or N-sided polygons. This is an important point to remember, especially when constructing your own mesh data and manually placing it into the vertex and index buffers of a D3DXMesh object.

Like the vertex buffer, the mesh index buffer is a standard IDirect3DIndexBuffer9 index buffer, as seen in previous lessons. Thus it can be accessed and manipulated in a similar fashion. All ID3DXBaseMesh derived interfaces inherit the LockIndexBuffer and UnlockIndexBuffer methods from the base class. This provides direct access to the mesh's underlying index data.

`HRESULT ID3DXMesh::LockIndexBuffer(DWORD Flags, VOID **ppData)`

DWORD Flags
These are the standard `D3DLOCK` flags that we have used when locking index buffers in previous lessons. The flags include `D3DLOCK_DISCARD`, `D3DLOCK_NOOVERWRITE`, `D3DLOCK_NOSYSLOCK`, `D3DLOCK_READONLY` and `NO_DIRTY_UPDATE` and can be used in combination to lock the buffer in an efficient manner.

VOID ** ppData
This is the address of a pointer that will point to the index data if the lock is successful.

We release the lock on a mesh index buffer with a single call to ID3DXMesh::UnlockIndexBuffer.

`HRESULT ID3DXMesh::UnlockIndexBuffer(VOID)`

The following code demonstrates how to lock and unlock a mesh index buffer. It is assumed in that pd3dxMesh is a pointer to an already initialized ID3DXMesh.

```
ID3DXMesh *pd3dxMesh;

// Pretend that we create the mesh here

WORD  *pIndices16;
DWORD *pIndices32;

if ( pd3dxMesh->GetOptions() & D3DXMESH_32BIT)
{
        // Lock the index buffer and get a pointer to the vertex data
        pd3dxMesh->LockIndexBuffer( 0, &pIndices32 );

        // This is where you could fill the index buffer using the pointer

        // When we are done we must remember to unlock
        pd3dxMesh->UnlockIndexBuffer();
}
else
{
        // Lock the index buffer and get a pointer to the vertex data
        pd3dxMesh-> LockIndexBuffer( 0, &pIndices16);

        // This is where you could fill the index buffer using the pointer

        // When we are done we must remember to unlock
        pd3dxMesh-> UnlockIndexBuffer();
}
```

In the above code, a call to ID3DXMesh::GetOptions is used to determine whether the mesh index buffer uses 32-bit or 16-bit indices. We will see in a moment that we will specify a number of option flags that inform the mesh creation and loading functions about the properties we would like our mesh buffers to exhibit. These properties include which memory pools we would like to use for our vertex/index buffers or whether the index buffer should contain 16-bit or 32-bit indices. The ID3DXMesh::GetOptions method allows us to retrieve the flags that were used to create the buffer. The return value is a bit-set stored in a DWORD that can be tested against a number of flags.

8.2.3 The Adjacency Buffer

To perform optimization and/or LOD on a mesh, it is necessary to know some information about the connectivity of the faces. That is, we wish to know which faces neighbor other faces. For example, LOD is performed by ‘collapsing’ two or more triangles into a single triangle. In order for this to be possible, the information must be at hand that tells the mesh how faces are connected to other faces. As a face in the context of any of the D3DX mesh types is always a triangle, and since a triangle always has three edges, we know that a single face can at most be adjacent to three other triangles. Therefore, when we need to send adjacency information to the mesh to perform one function or another, we will pass in an array with three DWORDs for each face. These values describe the face index numbers for neighboring

faces. If we have a mesh with 10 faces, then the adjacency information required would be an array of 30 DWORDs.

Fig 8.1 depicts a mesh with eight faces and the associated adjacency array. Keep in mind that while a triangle may be connected along its three edges to at most three other triangles, this does not mean that every triangle will be connected to three other faces. A mesh that contains a single triangle for example would obviously have no adjacent neighbors. Nevertheless, whether each face in the mesh is connected to three other faces or not, we still store three DWORDs per face. If a triangle edge is not connected to any other triangle in the mesh, then we will use a value of 0xFFFFFFFF to indicate this.

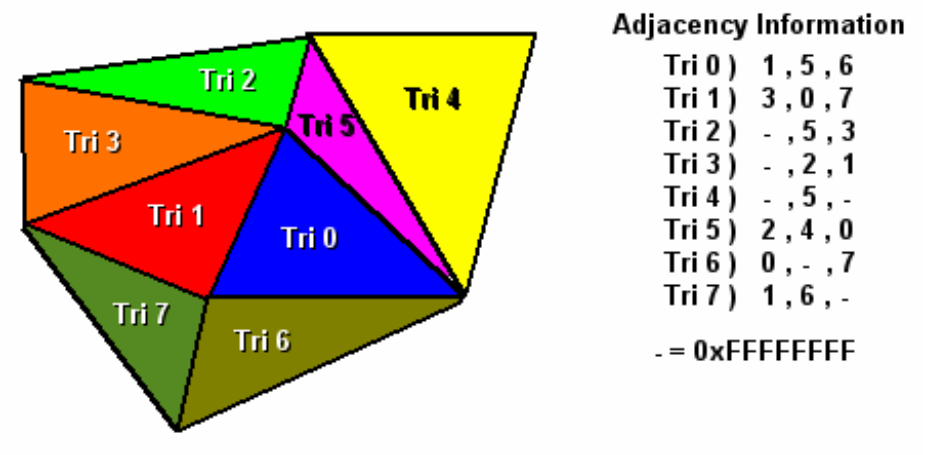


Figure 8.1

Looking at the list in Fig 8.1 we can see that triangle 0 is connected to faces 1, 5, and 6. Triangle 3 is only connected to two other triangles: 1 and 2. Using this adjacency information, a simplification process might decide to collapse triangles 0, 1, 6, and 7 by removing the shared middle vertex and collapsing the four triangles into two triangles.

While calculating face adjacency is not very difficult to do, we are spared even that minor hassle by D3DX. All mesh types derived from the ID3DXBaseMesh interface inherit a function called ID3DXBaseMesh::GenerateAdjacency to generate face adjacency data. To minimize per-mesh memory requirements, D3DX meshes do not generate this information automatically when meshes are created, so we must explicitly call this procedure if we require this data. Since we often require the use of adjacency data only once when optimizing a mesh, there is little point in keeping it in memory after the fact. For progressive meshes however, we may decide to keep this information handy as we will see later.

```
HRESULT ID3DXMesh::GenerateAdjacency(FLOAT fEpsilon,
                                     DWORD*pAdjacency);
```

FLOAT fEpsilon

Sometimes modeling packages create meshes such that faces that are supposed to be adjacent are not precisely so. This can happen for a number of reasons. For example, the level designer may have not used a correct alignment, leaving miniscule but significant gaps between faces. Sometimes this can be due to floating point rounding errors caused by cumulative vertex processing done on the mesh (such as

a recursive CSG process). Further, perhaps the two faces are not actually supposed to be touching, but you would like LOD and optimization functions to treat them as neighbors anyway. Fig 8.2 shows two triangles which should probably be connected, but in fact have a small gap between them. We can see immediately that the triangles would share no adjacent edges.

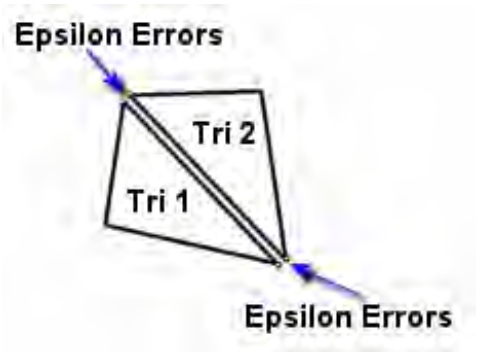


Figure 8.2

It can be frustrating when these tiny gaps prevent proper optimization or simplification. The fEpsilon value allows us to control how sensitive the GenerateAdjacency function is when these gaps are encountered. It is used when comparing vertices between neighboring triangles to test whether they are considered to be on the same edge, and therefore exist in a neighboring face. This is no different than testing floating point numbers or 3D vectors using an epsilon value. It simply allows us to configure the function to be tolerant about floating point errors. The larger the epsilon values, the more leeway will be given when comparing vertices and the more likely they are to be considered the same.

DWORD *pAdjacency
The second parameter is a pointer to a pre-allocated DWORD array large enough to hold three DWORDS for every face in the mesh.

The following code shows how we might generate the adjacency information for an already created mesh.

```
// Allocate adjacency buffer
DWORD *pAdjacency = new DWORD[pd3dxMesh->GetNumFaces() * 3];

// Generate adjacency with a 0.001 tolerance
pd3dxMesh->GenerateAdjacency( 0.001 , pAdjacency );
```

8.2.4 The Attribute Buffer

In Graphics Programming Module I we looked at how to assign textures and materials to our model faces. Recall that for each face we stored indices into global arrays that contained scene materials and textures. This is how we mapped faces in our mesh to textures and materials stored elsewhere in the application. During rendering, we would loop through each texture used by the mesh and render only the

faces that used that texture in a single draw primitive call. We would repeat this process for each texture used by the mesh until all faces had been rendered. D3DX mesh objects also provide this batch rendering technique which we know is so essential for good rendering performance.

Since D3DX meshes store data in vertex and index buffers and include no explicit face type per se, it becomes necessary to organize those buffers such that it is known in advance which groups of indices map to particular textures or materials. As we can no longer store texture and material indices in our mesh faces (because a mesh face is now just a collection of three indices in the mesh's index buffer) another buffer, called an *attribute buffer*, provides the means for doing so.

Whenever a D3DXMesh is created (either manually or via a call to the D3DXLoadMeshFromX function), an attribute buffer is created alongside the standard vertex and index buffers. There is one DWORD entry in this attribute buffer for every triangle in the index buffer. Each attribute buffer entry describes the Attribute ID for that face. All faces that share the same Attribute ID are said to belong to the same *subset* and are understood to require the same device states to be rendered. Therefore, all triangles in a subset can be rendered with a single draw primitive function call to minimize device state changes. All faces that belong to the same subset are not necessarily arranged in the index buffer in any particular order (although they can be, as we will soon discuss), but they will still be rendered together. Note that the mesh object itself does not contain any texture or material information; Attribute IDs provide the only potential link to such external concepts. In short, Attribute IDs provide a means to inform D3DX that faces with like properties can be rendered in a single draw call.

When building a mesh ourselves, the Attribute ID can describe anything we would like. It might be the index of a material or texture in a global array, or even an index into an array of structures that describe a combination of material, texture, and possibly even the lights used by the subset. Ultimately this ID is just a way for the application to inform the ID3DXMesh rendering function which faces belong to the same group and should be rendered together. It is important to understand that the application is still responsible for managing the assets that the face attributes map to (texture, materials, etc.) and for setting up the appropriate device states before rendering the subset.

Fig 8.3 depicts a nine triangle mesh, where each triangle uses one of five different textures. When the ID3DXMesh is created, there are empty vertex, index, and attribute buffers. The mesh itself contains no texture information. The textures used by the mesh are stored in a global texture array managed by the application. This is similar to the approach we took in our lab projects that used IWF files in earlier lessons -- we extracted the texture names from the file, loaded the textures into a global array, and stored the texture index in the face structure. We can no longer store the texture index in the face structure because the ID3DXMesh has no such concept. However we can store the texture index for each face in the attribute buffer instead.

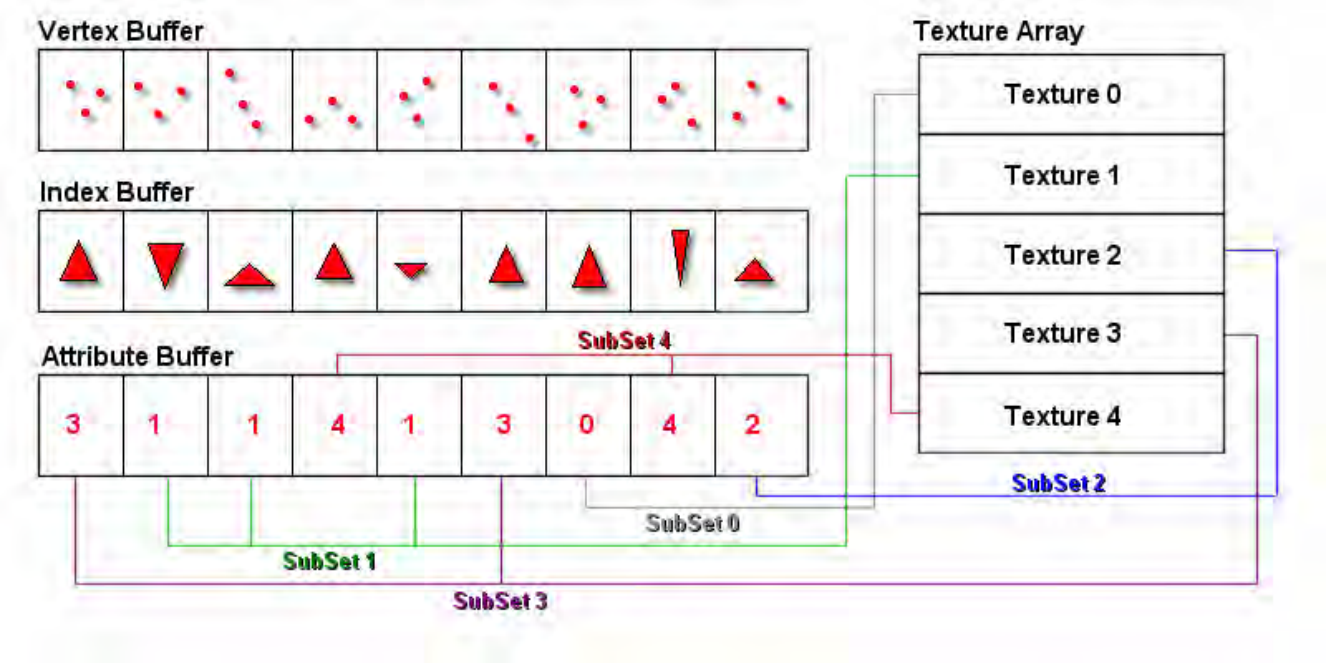


Figure 8.3

In this example, the ID3DXMesh itself has no knowledge that the Attribute IDs are in fact texture indices, but it does know that all faces with a matching Attribute ID are considered part of the same subset and can all be rendered together.

Fig 8.3 depicts the relationship between the triangle data stored in the index buffer and the Attribute ID stored in the attribute buffer. The 7th triangle uses texture 0, the 8th triangle uses texture 2, the 1st and 6th triangles use texture 3, the 2nd, 3rd, and 5th triangles use texture 1, and so on. Even in this simple example the mesh would benefit from attribute batching, as the faces belonging to a particular subset are fairly spread out inside the index buffer.

In a short while we will examine the D3DXLoadMeshFromX function. When the mesh is created using this function, its vertex, index, and attribute buffers are automatically filled with the correct data. To determine which Attribute IDs map to which asset types, the function returns an array of D3DXMATERIAL structures which contain a texture and material used by a given subset as specified by the X file that was loaded. So when we render subset 0 of the mesh, we set the texture and the material stored in the first element of the D3DXMATERIAL array. When rendering the second subset we set the texture and material stored in the second element of the D3DXMATERIAL array, and so on for each subset. Therefore, while the mesh itself does not maintain texture and material data, the D3DXLoadMesh… family of functions does return this information. The loading function correctly builds the attribute buffer such that each Attribute ID indexes into this D3DXMATERIAL array.

When loading an X file, the mesh and its internal buffers are created and filled automatically. While we rarely need to lock any of the mesh's internal buffers under these circumstances, when creating an ID3DXMesh object manually, we definitely need to be able to lock and unlock the attribute buffer.

The ID3DXMesh::LockAttributeBuffer and the ID3DXMesh::UnlockAttributeBuffer are not inherited from the ID3DXBaseMesh interface. Instead these are two of the new functions added to the ID3DXMesh interface beyond its inherited function set.

HRESULT LockAttributeBuffer(DWORD *Flags*, DWORD *ppData*);**

When locking the attribute buffer, we pass in one or more of the standard buffer locking flags to optimize the locking procedure. The second parameter is the address of a DWORD pointer which will point to the beginning of the attribute data on successful function return.

Unlocking the attribute buffer is a simple matter of calling the following function:

HRESULT UnlockAttributeBuffer(VOID);

8.2.5 Subset Rendering

To render a given subset in a mesh, we call ID3DXMesh::DrawSubset and pass in an Attribute ID. This function will iterate over the faces in the index buffer and render only those that match the ID passed in. Therefore, rendering a mesh in its entirety adds up to nothing more than looping through each subset defined in the mesh, setting the correct device states for that subset (texture and material, etc.) and then rendering the subset with a call to the ID3DXMesh::DrawSubset method. More efficient rendering is made possible when the subset data is grouped together in the mesh vertex and index buffers. We will discuss mesh optimization in detail a little later in the chapter.

HRESULT DrawSubset(DWORD *AttribId*);

The mesh in Fig 8.3 could be rendered one subset at a time using the following code.

```
for(int I = 0; I < m_nNumberOfTextures; I++)
{
     pDevice->SetTexture(0, pTextureArray[I]);     // Set Subset Attribute
     pd3dxMesh->DrawSubset(I);                     // Render Subset
}
```

Again it should be noted that while this example is using the texture index as the Attribute ID for each subset, this ID could instead be the index of an arbitrary structure in an array which might hold much more per-face information: materials, transparency, multiple textures, lights, etc. used by that subset. All that matters to D3DX is that faces with the same Attribute ID share like states and can be batch rendered. What these Attribute IDs mean to the application is of no concern to the mesh object. It is the application that is responsible for setting the correct device states before rendering the subset that corresponds to the particular Attribute ID.

8.3 ID3DXBaseMesh Derived Functions

The purpose of most of the base mesh derived functions is pretty self-explanatory, but we will quickly review them just to get a feel for the information that a mesh stores and how to access it. Then in the next section, we will look at how to actually create and use a mesh.

```
HRESULT GetDevice( LPDIRECT3DDEVICE9 *ppDevice );
```

When we create a mesh using the D3DXCreateMesh function or the D3DXLoadMesh… family of functions, we must specify a pointer to the device interface through which the mesh will be created. This is required because the device owns the vertex buffer and index buffer memory that is allocated, and thus the mesh object is essentially owned by the device as well. The ID3DXBaseMesh::GetDevice function allows us to retrieve the device that owns the mesh.

```
DWORD GetFVF(VOID);
```

When we create a mesh using the D3DXCreateMeshFVF function, we directly specify the flexible vertex format flags for the mesh vertex buffer. The ID3DXBaseMesh::GetFVF function allows us to retrieve this information so that we know the layout of the vertices stored in this internal vertex buffer.

When using the D3DXLoadMesh… family of functions, being able to retrieve the FVF flags for the mesh vertex structure is important because we often have no direct control over the original vertex format the mesh was created with. For example, when using the D3DXLoadMeshFromX function, the function will choose a vertex format that most closely matches the vertex information stored in the X file. If the vertices in the X file contain vertex normals and three sets of texture coordinates, the mesh vertex buffer will automatically be created to accommodate this information. Therefore, when the mesh is created from an X file, we need this function to determine the format with which the vertex buffer was initialized. You will see later when we discuss mesh cloning that we are not restricted to using vertices in the format specified in the X file. We can clone the mesh and specify a different vertex format to better suit the needs of our application.

```
HRESULT GetIndexBuffer( LPDIRECT3DINDEXBUFFER9 *ppIB );
```

The ID3DXBaseMesh::GetIndexBuffer function returns a pointer to the IDirect3DIndexBuffer9 interface for the internal mesh index buffer. Retrieving the index buffer interface (or the vertex buffer interface) can be useful if we wish to render the mesh data manually and not use the DrawSubset functionality. If you had your own custom rendering code that you needed to use and merely wanted to use the ID3DXMesh to optimize your dataset, this would be your means for accessing the indices. After the data was optimized, you would simply retrieve the vertex and index buffer interfaces and use them as you would under normal circumstances.

```
HRESULT GetVertexBuffer( LPDIRECT3DVERTEXBUFFER9 *ppVB );
```

Like the previous function, this function returns an interface pointer to the mesh vertex buffer so that you can use it for custom rendering or other application required manipulation.

```
DWORD GetNumBytesPerVertex(VOID);
```

This function returns the size (in bytes) of the vertex structure used by this mesh. This is useful when you need to render the mesh data manually since you need to specify the stride of the vertex when binding the vertex buffer with the IDirect3DDevice9::SetStreamSource function.

```
DWORD GetNumFaces(VOID);
```

This function returns the number of faces (triangles) stored in the mesh index buffer. Since the mesh always stores its geometry as an indexed triangle list, the result of this function will be the total number of indices in the mesh index buffer divided by 3.

```
DWORD GetNumVertices(VOID);
```

This function returns the number of vertices in the mesh vertex buffer.

```
DWORD GetOptions(VOID);
```

This function returns a DWORD which stores a series of flags that were used during creation of the mesh. The flags include information about the memory pools used for the vertex and index buffers, the size of the indices (32-bit or 16-bit) in the index buffer, whether the vertex or index buffer have been created as dynamic buffers, and whether or not they are write-only buffers.

8.4 Mesh Optimization

There is one more topic to discuss before we finally look at how to create/load a mesh: optimization of vertex and index data. While this may seem like a strange way round to do things, it should help us better understand exactly what information the D3DXLoadMeshFromX function is returning to us as well as how to efficiently organize our data when we create meshes manually.

Whether or not you intend to use the ID3DXMesh (or any of its sibling interfaces) for your own mesh storage or rendering, it would be a mistake to overlook the geometry optimization features offered by the ID3DXMesh::Optimize and the ID3DXMesh::OptimizeInPlace functions we are about to examine. Even if you have your own mesh containers and rendering API all worked out, your application may well benefit from temporarily loading the mesh data into an ID3DXMesh and using its optimization functions before copying the optimized data back into your proprietary structures. This can save you some time and energy developing such optimization routines yourself.

It is a sad but true fact that if you are starting off as a hobbyist game developer, you will likely not have access to artists to develop 3D models specifically for your applications. Often you will be forced to use models that you find on the Internet or other places where they may or may not be stored in the file in an optimal rendering arrangement. The Optimize and OptimizeInPlace functions are both introduced in the ID3DXMesh interface and will provide some relief. Both functions perform identical optimizations to the mesh data and, for the most part, take identical parameter lists. The difference between them is that the Optimize function generates a new mesh containing the optimized data and the original mesh

remains intact. The OptimizeInPlace function does not create or return a new mesh, it directly optimizes the data stored in the current mesh object. Let us take a look at the OptimizeInPlace function first.

```
HRESULT OptimizeInPlace( DWORD Flags,
                         CONST DWORD *pAdjacencyIn,
                         DWORD *pAdjacencyOut,
                         DWORD *pFaceRemap,
                         LPD3DXBUFFER *ppVertexRemap
                        );
```

DWORD Flags

The Flags parameter describes the type of optimization we would like to perform. We typically choose only one of the standard optimization flags, but we can combine the standard optimization flags with additional modifier flags. The standard optimization flags are ordered and cumulative. That is, each one listed also contains the flags for the optimizations that precede it.

D3DXMESHOPT_COMPACT – When specifying this flag, the optimizer removes vertices and/or indices that are not required. A classic example of this would be stray vertices in the vertex buffer that are never referenced by any indices in the index buffer. It is also possible that the geometry from which the mesh was created contains degenerate triangles. Because meshes are always stored as triangle lists and degenerate triangles are of little use in these situations, they will also be removed. This is the base optimization method that is performed by all other optimization flags. It is also the quickest and cheapest optimization that we can perform. Note that it may involve index and vertex data reshuffling or reordering in order to fulfill the compaction requirements.

D3DXMESHOPT_ATTRSORT - The attribute sort optimization is performed in addition to the compaction optimization listed above. That is, you do not have to specify both `D3DXMESHOPT_COMPACT` and `D3DXMESHOPT_ATTRSORT` since the `D3DXMESHOPT_ATTRSORT` flag by itself will cause a compaction and an attribute sort optimization to be performed. When a mesh is first created, either in code or by the D3DXLoadMeshFromX function, faces are typically added to the index buffer as they are encountered. This means that we may have faces that share the same Attribute ID (and are therefore part of the same subset) randomly dispersed within the index buffer. The optimizer sorts the data stored within the mesh so that all the faces that share the same Attribute ID are consecutive within the index buffer. This allows them to be efficiently batch rendered.

Fig 8.4 shows how a D3DX mesh might be created. In this example we shall assume that we have loaded geometry from an IWF file and stored it inside a D3DXMesh. This would be a simple case of calling D3DXCreateMesh to create the empty mesh object and then locking the vertex and index buffers and copying in the IWF data (we discussed how easy it was to load IWF files into memory with the IWF SDK in Chapter Five). Once we have the data file in memory, we loop through each face that was loaded (temporarily stored in the IWFSurface vector) and copy the vertices of the face into the vertex buffer and the indices of the face into the index buffer (arranged as indexed triangle lists).

To keep this example simple, we have used only the render material referenced by the surface as the Attribute ID for the face. Thus all faces that use the same material in the IWF file are said to belong to the same subset. The point here is not how the data gets into the mesh, but that when we add data in an arbitrary order to the internal buffers, triangles belonging to the same subset may not be consecutive in the buffers. In this example, we added nine triangles to the mesh in an arbitrary order; the order that they were encountered in the external file. When we add each triangle, we add its three vertices to the vertex buffer, its indices to the index buffer, and the triangle Attribute ID (the material index) to the attribute buffer.

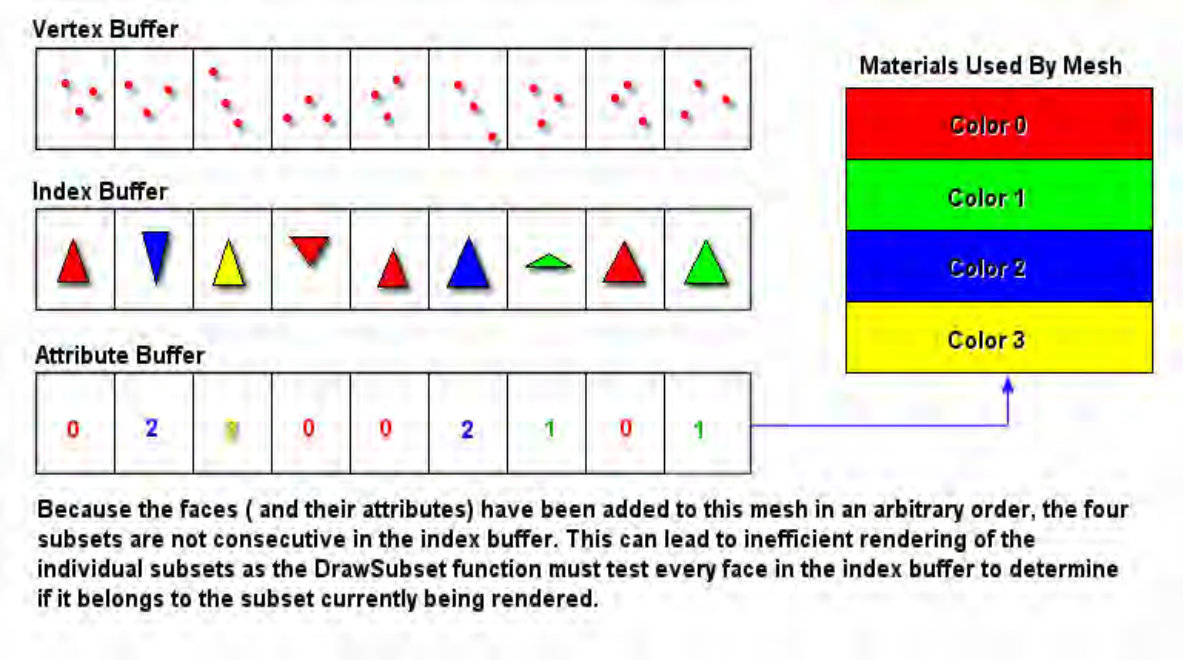


Figure 8.4

Given the disorganized nature of the buffers in Fig 8.4, if we were to call the ID3DXMesh::DrawSubset method and pass in an ID of 0, the function would need to render the 1st, 4th, 5th, and 8th triangles. Because the mesh cannot assume that all triangles belonging to the same subset are batched together by default, when we issue the render call for a subset, the function must loop through the entire attribute buffer to find and render all triangles that match the selected Attribute ID one at a time.

When we use the `D3DXOPTMESH_ATTRSORT` flag, the vertex and index buffers will be reordered such that vertices and indices belonging to the same subsets are batched together in their respective buffers. The function internally builds an *attribute table* that describes where a subset's faces begin and end in the index buffer and where the subset's vertices begin and end in the vertex buffer. This sets the stage for efficient DrawIndexedPrimitive calls.

Fig 8.5 shows how the mesh might look after an attribute optimization has been performed. Study the following image and see if you notice anything strange about the attribute buffer.

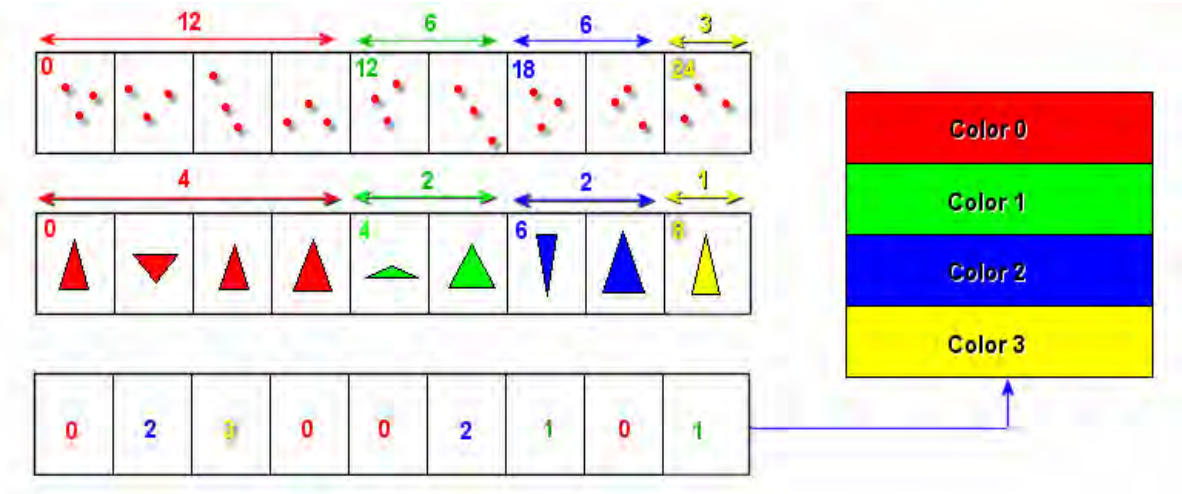


Figure 8.5

Notice how the attribute buffer no longer correctly maps to the faces. When we perform an attribute sort on a mesh, the mesh is flagged internally as having been attribute sorted and an attribute table is built. From that point on, the attribute buffer is no longer used in any calls to the DrawSubset function. Instead, batch rendering information is pulled from the from the attribute table. The DrawSubset function no longer needs to loop through the attribute buffer for each subset and find the matching triangles, because the attribute table describes where a subset begins and ends in the index and vertex buffers.

We can see in Fig 8.5 that the attribute sort has correctly batched all faces belonging to the same subset in the index and vertex buffers. It also shows how subset 0 starts at face 0 in the index buffer and consists of four triangles. The vertices for subset 0 begin at vertex 0 in the vertex buffer and the range consists of 12 vertices. Now the DrawSubset function can quickly jump to the beginning of the subset in the vertex and index buffers and pump them into the pipeline with a single DrawIndexedPrimitive call.

Once a mesh has been attribute sorted, we can gain access to the internal attribute table using the ID3DXBaseMesh::GetAttributeTable method. This returns an array of `D3DXATTRIBUTERANGE` structures (one per subset) describing where a given subset begins and ends in the vertex and index buffers. If you intend to perform geometry or attribute manipulation on an already optimized mesh, you may want to update the attribute table with the new information rather than calling the Optimize function again.

The `D3DXATTRIBUTERANGE` range structure is shown below.

```
typedef struct _D3DXATTRIBUTERANGE {
    DWORD AttribId;
    DWORD FaceStart;
    DWORD FaceCount;
    DWORD VertexStart;
    DWORD VertexCount;
} D3DXATTRIBUTERANGE;
```

The first member of the structure contains the zero-based Attribute ID for the subset. If this value was set to 5 for example, then this structure would describe where the vertices and indices for subset 5 begin and end in the vertex and index buffers. The FaceStart member holds the triangle number (offset from zero) where the subset begins in the index buffer. The FaceCount member describes how many triangles beginning at FaceStart belong to the subset. The VertexStart member describes the zero-based index of the vertex in the vertex buffer where the subset vertices begin. The VertexCount member describes the number of vertices that belong to the subset. Essentially, these are the members you would use if you decided to call DrawIndexedPrimitive yourself.

To retrieve the attribute table, we use the ID3DXBaseMesh inherited function GetAttributeTable.

```
HRESULT GetAttributeTable
(
    D3DXATTRIBUTERANGE *pAttribTable,
    DWORD *pAttribTableSize
);
```

The first parameter is a pointer to a pre-allocated array of `D3DXATTRIBUTERANGE` structures that the function will fill with subset information. The second parameter is the number of subsets (starting at zero) we would like to retrieve information for. We must make sure that we have allocated at least as many `D3DXATTRIBUTERANGE` structures as the number specified in the pAttribTableSize parameter.

If we would like all of the information for every subset the mesh contains, we must first know how many `D3DXATTRIBUTERANGE` structures to allocate memory for. If we call this function and set the first parameter to NULL and pass the address of a DWORD as the second parameter, the function will fill the passed DWORD with the number of subsets in the mesh. This allows us to allocate the memory for the `D3DXATTRIBUTERANGE` structures before issuing a second call to the function to retrieve the actual attribute table. It should be noted that an attribute table does not exist until the mesh has undergone an attribute sort. If you call this function without first attribute sort optimizing the mesh, 0 will be returned in the pAttributeTableSize parameter.

The following code snippet demonstrates how we might optimize a mesh such that it is compacted and attribute sorted. We then retrieve the mesh attribute table. The code assumes that pd3dxMesh has already been created and filled with data but has not yet been optimized.

```
// Allocate adjacency buffer needed for optimize
DWORD *pAdjacency = new DWORD[ pd3dxMesh->GetNumFaces() * 3];

// Generate adjacency with a 0.001 tolerance
pd3dxMesh->GenerateAdjacency( 0.001 , pAdjacency );

// Optimize the mesh ( D3DXOPTMESH_ATTRSORT = Attribute Sort + Compact )
pd3dxMesh->OptimizeInPlace(D3DXOPTMESH_ATTRSORT,pAdjacency,NULL,NULL,NULL);

// No longer need the adjacency information
delete []pAdjacency;
```

```
// We need to find out how many subsets are in this optimized mesh
DWORD NumberOfAttributes;

// Get the number of subsets
pd3dxMesh->GetAttributeTable( NULL , &NumberOfAttributes );

// Allocate enough D3DXATTRIBUTERANGE structures, one for each subset
D3DXATTRIBUTERANGE *pAttrRange = new D3DXATTRIBUTERANGE[NumberOfAttributes];

// Call the function again to populate array with the subset information.
pd3dxMesh->GetAttributeTable( pAttrRange , &NumberOfAttributes );

// Examine the actual attribute range data here and do what you want with it
```

Notice that in order to perform an optimization of any kind, we must supply the function with face adjacency information.

The ID3DXMesh interface includes an additional function (beyond the ID3DXBaseMesh inherited functions) that can be used to specify an attribute table manually.

HRESULT SetAttributeTable(CONST D3DXATTRIBUTERANGE *_pAttrTable_,
DWORD _cAttrTableSize_);

The first parameter is the new array of `D3DXATTRIBUTERANGE` structures and the second parameter is the number of attributes in the array. It is very rare that you will need to modify the attribute table of an optimized mesh, but if necessary, then this is the function you should use to set the new data. Note that it does not alter the index and vertex buffer information in any way; it simply sets the attribute table internally. When a mesh has been attribute sorted, it is marked as having been so.

> **Note:** When you set the attribute table manually you must make sure that you correctly identify the subset ranges, because the DrawSubset function will blindly use this information to render each subset. When we call DrawSubset for example and pass in an attribute ID of 5, the subset's vertex and index start and count values will be taken from this table regardless of what is stored in the attribute buffer.

D3DXMESHOPT_VERTEXCACHE – This flag is used to inform the optimizer that we would like the vertex and index buffers of the mesh to be optimized to better utilize the vertex cache available on most modern hardware accelerated 3D graphics cards. This will often involve reordering the vertex and index data of the mesh such that vertices that are transformed and stored in the vertex cache on the GPU are reused as much as possible. If a vertex is shared by 10 faces for example, this optimization will try to order the index buffer and vertex buffer such that this vertex is only transformed and lit once. It can then render all faces that use the vertex before it is evicted from the vertex cache. This is an extremely effective optimization which aims to increase the cache hit rate of hardware vertex caches.

Again, specifying this single flag also includes the optimizations performed by the two previous flags just discussed. Therefore, specifying `D3DXMESHOPT_VERTEXCACHE` also performs the `D3DMESHOPT_COMPACT` and `D3DXMESHOPT_ATTRSORT` optimization processes. The `D3DXMESHOPT_VERTEXCACHE` flag and the `D3DMESHOPT_STRIPREORDER` flags (discussed next) are mutually exclusive since they have very different goals.

D3DXMESHOPT_STRIPREORDER – The name of this optimization flag can sometimes cause some confusion. It seems to imply that the vertex and index buffer data would be modified to be rendered as indexed triangle strips, but this is not the case. A D3DXMesh will always be stored using an indexed triangle list format. So what does this optimization actually do?

Some 3D graphics cards can only render triangle strips at the hardware level. This means that when we render triangle lists or fans for example, the driver will convert these primitive types to strips before rendering. This is not something we are ever aware of or have to make provisions for, but this strip creation process can carry overhead on such cards. While the overhead is generally minimal, this optimization flag makes it clear that we would like the indexed triangle lists of the mesh to be re-arranged such that they can be more quickly be converted into strips by the driver. The optimizer reorders the faces so that, as much as possible, physically adjacent triangles are consecutively referenced. This increases the length of adjacent triangle runs in the buffer, which helps speed things up in cases where strip generation is required.

This flag and `D3DXMESHOPT_VERTEXCACHE` are mutually exclusive due to the fact that they both try to attain a best-fit ordering given a different set of rules. The `D3DOPTMESH_STRIPREORDER` flag, like the `D3DMESHOPT_VERTEXCACHE` flag, contains all the optimization processes previously described.

The optimization flags can thus be summed up as follows:

```
D3DXMESHOPT_COMPACT      =    Compact process only.
D3DXMESHOPT_ATTRSORT     =    Compact + Attribute Sort.
D3DXMESHOPT_VERTEXCACHE  =    Compact + Attribute Sort + Cache Reorder.
D3DXMESHOPT_STRIPREORDER =    Compact + Attribute Sort + Strip Reorder.
```

Modifier Flags

There are a number of modifier flags that can be combined with any of the previously discussed standalone optimization flags. They are listed below along with a brief description of their purpose.

D3DXMESHOPT_IGNOREVERTS - This flag instructs the optimizer not to touch the vertices and to work only with the face/index data. Thus, if we were to perform a compact optimization with this modifier flag, no vertices would be removed -- even if they were never referenced. In that case, only unused face indices would be removed. This is a useful modifier flag if you wish to optimize the mesh but want the vertex buffer to remain unchanged.

D3DXMESHOPT_DONOTSPLIT - This is a modifier flag that is used primarily for attribute sorting. When an attribute sort optimization is performed, vertices that are used by multiple attribute groups may be split (i.e. duplicated) so that a best-order scenario is obtained when batch rendering, while keeping vertices in a local neighborhood to the subset. This way, software vertex processing performance is not compromised.

For example, let us say that we had 300 attributes (subsets) and 5000 vertices. Assume that attribute 0 and attribute 299 both reference vertex 4999. In this case, the vertex attribute range for attribute 0 would span the entire contents of the vertex buffer. This would seriously hurt

software vertex processing performance since we know that when calling DrawIndexedPrimitive on a software vertex processing device, the entire range of specified vertices is transformed in one pass. In this example then, attribute 0 would have a vertex start index of 0 and a vertex count of 5000. So even if this was only a single triangle subset, all 5000 vertices would need to be transformed and possibly lit when rendering subset 0. This is not a problem when using a hardware vertex processing device because the vertex cache is used instead. So in order for mesh performance to downgrade gracefully to a software vertex processing device, under these conditions the vertex may be duplicated and moved to the beginning of the buffer. Attribute 0 would then have a localized set of vertices to minimize block vertex transformations on a software vertex processing device.

This duplication process is performed automatically by the attribute sort optimization process and normally that is a good thing. However, if for any reason you consider this a problem and wish to prevent the automatic duplication of vertices, specifying this flag will prevent this duplication optimization from being performed.

D3DXMESHOPT_DEVICEINDEPENDENT - During vertex cache optimization, a specific vertex cache size is used which coincides with the cache provided on the hardware. This approach works well for modern hardware. This flag specifies that the optimizing routine should alternatively assume a default, device-independent vertex cache size (a fixed cache size) because this usually works better on older hardware.

Example Combinations:

```
D3DXMESHOPT_VERTEXCACHE|D3DXMESHOPT_DEVICEINDEPENDANT|D3DXMESHOPT_DONOTSPLIT
```

With this combination, the data will be compacted to remove un-referenced vertices and degenerate triangles and will be attribute sorted. When the attribute sort is being performed, the optimization routine will not duplicate vertices to minimize subset vertex ranges. This would make the mesh transform and render much slower on a software vertex processing device. The mesh will also have its triangles and vertices reordered to better utilize the vertex cache. In this case we want a fixed, device-independent vertex cache size to be assumed.

```
D3DXMESHOPT_ATTRSORT|D3DXMESHOPT_IGNOREVERTS
```

This combination instructs the optimization routine to compact and attribute sort only the index data and not alter the vertex buffer in any way. If there are any un-referenced vertices in the vertex buffer, they will not be removed. Furthermore, since we are stating that we do not want the vertex data touched, vertices will not be duplicated to produce better localized vertices for a subset. This means that such a mesh will suffer the same poor software vertex processing performance if an attribute/subset references vertices over a large span of the vertex buffer.

CONST DWORD *pAdjacencyIn

This is a pointer to an adjacency array calculated using ID3DXBaseMesh::GenerateAdjacency. As discussed, this is an array of DWORDS (3 per face) describing how faces are connected. Although it might be argued that it would be nice if the Optimize and OptimizeInPlace functions called

GenerateAdjacency automatically, this would require that the adjacency information be recalculated with every call to the optimize functions. Since you may already have the adjacency information or need to call optimize several times for several copies of the mesh, you can just calculate the adjacency information once and pass it in each time it is needed.

DWORD *pAdjacencyOut

Most mesh optimizations involve the rearranging of vertices and indices in the internal buffers. Since it is possible that faces may be removed by the compaction algorithm or rearranged by the attribute sort/ vertex cache optimizations, the adjacency index information has likely changed as well. If your application needs to maintain the adjacency data, then this pointer can be filled with the new adjacency information when the function returns. If we pass NULL, the adjacency information will not be returned. Thus if you need the adjacency information in the future, you will need to call the GenerateAdjacency function again.

Since the optimization process never introduces new faces (at most it only removes triangles), as long as this buffer is at least as large as the original adjacency buffer (pAdjacencyIn), there will be enough room to house the adjacency information of the optimized mesh.

DWORD *pFaceRemap

This parameter allows us to pass in a pointer to a pre-allocated array that on function return will contain information describing how faces have been moved around inside the index buffer after the optimization step. This array is useful when you have external attributes/properties associated with the mesh triangles that need to be updated after the mesh is optimized. On function return, the following relationship will be true:

pFaceRemap[***OldFaceIndex***] = ***NewFaceIndex***

For example, imagine that you were writing a level editing system and that you have several decals in your levels that are attached to faces by an index. When the mesh is optimized, the faces may have been rearranged such that each face now has a completely different index. In such a case, you would want to update the face index stored in your decal structure so that it tracked the faces it was attached to after the optimization process. If the decal was attached to face 5 for example, after the optimization face 5 may have been moved to face slot 10 in the index buffer. The decal would need to be updated so that it did not blindly attach itself to face index 5, which would now be a completely different face.

Before calling the optimization function, you can inquire how many faces the mesh has and allocate an array of DWORDS (1 per face) and pass this buffer into the function. When the function returns, each element in the array will describe the new post-optimization index for each face. In our example we said that the decal was attached originally to face 5. When the optimization was performed, this face had been moved to become face 10 in the index buffer. In this case, we could simply check the value stored in pFaceRemap[5] and it would hold a value of 10 -- the new location of the original face 5. We could then update the face index stored in the decal.

If you do not require this information about the post optimized mesh (which is often the case), you can set this parameter to NULL and no face remap information will be returned.

LPD3DXBUFFER *pVertexRemap

The final parameter is the address of a pointer to an ID3DXBuffer interface. In principle, it serves the same purpose for vertices as the previous parameter did for faces. The ID3DXBuffer interface provides access to generic memory buffers that are used by various D3DX functions to return or accept different data types (vertex or adjacency information for example). Unlike the face remap parameter where we pre-allocated the array, the vertex remap parameter should be the address of a pointer only. This will be used to allocate the buffer inside the function and return the information to the caller. If your application does not need the returned vertex buffer re-map information, you can just pass NULL. But if you do use it, make sure to release the ID3DXBuffer interface to free the underlying memory.

If we do not specify the `D3DMESHOPT_IGNOREVERTS` modifier flag, it is likely that the vertex data was reorganized during the call. Therefore, we will get back a buffer that contains enough space to hold one DWORD for every vertex in the original pre-optimized mesh. For each element in the returned buffer the following relationship is true.

pVertexRemap [***OldVertexIndex***] = ***NewVertexIndex***

Optimization Example I

We have now covered all of the parameters to the ID3DXMesh::OptimizeInPlace function. We have seen that the optimization features are quite impressive and extremely easy to use. Most of the time we will not need the face re-map and vertex re-map information or the newly compiled adjacency information and we will pass NULL as the last three parameters. However, the following code shows how we might optimize the mesh and store all returned information just in case. This code assumes that pd3dxMesh is a pointer to a valid ID3DXMesh interface.

```
// Allocate adjacency buffer needed for optimize
DWORD *pOldAdjacency = new DWORD[ pd3dxMesh->GetNumFaces() * 3];
DWORD *pNewAdjacency = new DWORD[ pd3dxMesh->GetNumFaces() * 3];
DWORD *pFaceRemap    = new DWORD[ pd3dxMesh->GetNumFaces()    ];
ID3DXBuffer* pVertexRemap;

// Generate adjacency with a 0.001 tolerance
pd3dxMesh->GenerateAdjacency( 0.001 , &pOldAdjacency );

// Optimize the mesh ( D3DXOPTMESH_ATTRSORT = Attribute Sort + Compact )
pd3dxMesh->OptimizeInPlace ( D3DXMESHOPT_ATTRSORT, pOldAdjacency,
                             pNewAdjacency, pFaceRemap, &pVertexRemap );
```

Optimization Example II

In the next example we do not need the face re-map, vertex re-map, or new adjacency information. This simplifies the code to only a few lines. Here we perform a compaction, attribute sort, and a vertex cache optimization.

```
// Allocate adjacency buffer needed for optimize
DWORD *pOldAdjacency = new DWORD [ pd3dxMesh->GetNumFaces() * 3];

// Generate adjacency with a 0.001 tolerance
pd3dxMesh->GenerateAdjacency( 0.001 , &pOldAdjacency );

// Optimize the mesh ( D3DXMESHOPT_VERTEXCACHE = Cache + Attribute Sort + Compact )
pd3dxMesh->OptimizeInPlace(D3DXMESHOPT_VERTEXCACHE, pOldAdjacency,
                           NULL, NULL, NULL );
```

The ID3DXMesh::Optimize function works in almost exactly the same way as the OptimizeInPlace call we just studied. The exception is that the current mesh data is not altered in any way. Instead, a new mesh is created and the optimization takes place there. Once the function has returned, you will have two copies of the mesh: the original un-optimized version whose interface you used to issue the ID3DXMesh::Optimize call, and a new optimized D3DXMesh object. These meshes have no interdependencies, so the original un-optimized mesh could be released if no longer needed.

```
HRESULT Optimize
(
    DWORD Flags,
    CONST DWORD *pAdjacencyIn,
    DWORD *pAdjacencyOut,
    DWORD *pFaceRemap,
    LPD3DXBUFFER *ppVertexRemap,
    LPD3DXMESH   *ppOptimizedMesh
);
```

Unless otherwise specified, when we use the ID3DXMesh::Optimize function, the newly generated mesh will inherit all of the creation parameters of the original mesh.

> **Note:** When we create (or clone) a mesh, we can specify one or more creation flags using members of the D3DXMESH enumerated type. This allows us to control things such as the resource pools the index and vertex buffers are created in. Because the Optimize function is creating a new mesh for the optimized data, we can also combine members of the D3DXMESH enumerated type with the usual D3DXMESHOPT flags discussed previously. This allows us to specify not only the optimization flags we require, but also the creation flags for the newly created optimized mesh.
>
> Although the D3DXMESH enumerated type will be covered shortly, you should know for now that we cannot use the D3DXMESH32_BIT, D3DXMESH_IB_WRITEONLY and D3DXMESH_WRITEONLY mesh creation flags during this call.

8.5 ID3DXBuffer

Before we cover the mesh loading routines, let us briefly discuss the ID3DXBuffer interface in a little more detail. The ID3DXBuffer is used to return arbitrary length data from D3DX functions. It is used in a number of function calls to return things like error message strings, face adjacency, and vertex and material information from optimization and loading functions.

Once a buffer of information has been returned from a D3DX function, you can save it for later use, retrieve a pointer to the buffer data area, or discard the buffer information by releasing the interface. Typically we will copy the data out to more useful structures used by our application. Just remember to release the buffer interface when it is no longer needed.

This interface is derived from IUnknown and exposes only two methods. One returns the size of the buffer and the other returns a void pointer to the raw buffer data. The two methods of the ID3DXBuffer interface are shown and described next.

```
DWORD ID3DXBuffer::GetBufferSize(VOID);
```

This function returns a DWORD describing the size of the buffer in bytes.

```
LPVOID ID3DXBuffer::GetBufferPointer(VOID);
```

This function returns a void pointer to the raw buffer data. This is similar to a pointer that gets returned when we lock a vertex buffer or a texture. Notice that there is no concept of locking or unlocking an ID3DXBuffer object like we saw with other resources.

To use the ID3DXBuffer interface for storing your own data collections, you can create an ID3DXBuffer object by calling the global D3DX function D3DXCreateBuffer.

```
HRESULT D3DXCreateBuffer(DWORD NumBytes, LPD3DXBUFFER *ppBuffer);
```

To allocate an ID3DXBuffer using this function, we pass in the size of the buffer to allocate and the address of an ID3DXBuffer interface pointer. If the function is successful, this second parameter will point to a valid ID3DXBuffer interface.

8.6 Mesh Loading Functions

8.6.1 D3DXLoadMeshFromX

The global D3DX function D3DXLoadMeshFromX creates a new D3DXMesh and loads X file data into its vertex, index, and attribute buffers. It basically creates a ready-to-render mesh for our application with a single function call. However, no data optimization will have been performed on the returned mesh, so if you want to optimize the resulting mesh, you will need to do this as a separate step with a call to ID3DXMesh::OptimizeInPlace after the mesh has been loaded. More often than not, you will be loading your meshes from X files. This function does all the hard work of parsing the X file data and creating a new mesh for you. It is shown below along with a description of its parameter list.

```
HRESULT D3DXLoadMeshFromX
(
    LPCTSTR pFilename,
    DWORD Options,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXBUFFER* ppAdjacency,
    LPD3DXBUFFER* ppMaterials,
    LPD3DXBUFFER* ppEffectInstances,
    DWORD* pNumMaterials,
    LPD3DXMESH* ppMesh
);
```

LPCTSTR pFilename

This is a string containing the file name of the X file we wish to load. If the compiler settings require Unicode, the data type LPCTSTR resolves to LPCWSTR. Otherwise, the string data type resolves to LPCSTR.

DWORD Options

Here we specify one or more flags from the `D3DXMESH` enumerated type describing how we would like the mesh to be created. This enumerated type contains members that can be used for all mesh types (including ID3DXPatchMesh, which is not covered in this chapter), so not all members will be valid for ID3DXMesh creation. These flags control items like the vertex and index buffer resource pool and whether the buffers are static or dynamic. These are basically wrappers around the standard vertex buffer and index buffer creation flags we discussed in Chapter Three.

D3DXMESH_32BIT

If this flag is specified then the index buffer will be created with 32-bit indices instead of the default 16-bit indices. This is analogous to specifying the `D3DFMT_INDEX32` usage flag when creating an index buffer using the IDirect3DDevice9::CreateIndexBuffer function.

D3DXMESH_DONOTCLIP

This informs the mesh to create its underlying vertex and index buffers such that they will never require clipping by the pipeline. This is analogous to the `D3DUSAGE_DONOTCLIP` flag used when manually creating a vertex buffer with the IDirect3DDevice9::CreateVertexBuffer function. The default state (the absence of this flag) is to create buffers that pass through the clipping process of the pipeline. If you know that your mesh will always be rendered such that it is within the bounds of the render target, this flag can speed up pipeline processing.

D3DXMESH_POINTS

This is a simple wrapper around creating a vertex buffer with the `D3DUSAGE_POINTS` flag. This is used for rendering a special primitive called a point sprite. Point sprites will be discussed in Module III.

D3DXMESH_RTPATCHES
D3DXMESH_NPATCHES

These flags can be used to create the mesh for storage of higher-order primitives and N-patches. These types are beyond the scope of this chapter and will be covered later in this course series. They are analogous to creating vertex buffers and index buffers manually using the `D3DUSAGE_RTPATCHES` and `D3DUSAGE_NPATCHES` flags.

D3DXMESH_VB_SYSTEMMEM
D3DXMESH_VB_MANAGED

These two flags are mutually exclusive. They allow us to specify the resource pool where we would like the mesh vertex buffer stored. By default, if we do not specify one of these flags, the vertex buffer will be created in the default memory pool. The same rules about resource persistence apply here when a device is lost and reset (see Chapter Three). Default pool buffers become invalid when a device is lost. This requires having to release and recreate the mesh from scratch using either the D3DXLoadMesh… or D3DXCreateMesh functions. To avoid this step when using the default pool, a common practice is to store two meshes: one in system memory and one in video memory. When the video memory copy becomes lost, it is released and

recreated by cloning from the system memory mesh. As discussed in Chapter Three, all of this can be avoided through the use of the managed memory pool (`D3DXMESH_VB_MANAGED`). When the device is lost and reset, the DirectX memory manager will automatically handle the release and recreation of the internal buffers.

D3DXMESH_VB_WRITEONLY
This is analogous to specifying the `D3DUSAGE_WRITEONLY` flag during normal vertex buffer creation. In Chapter Three we discussed using this flag for optimal rendering performance. However, we also learned that we should not do this if we intend to read from the vertex buffer since the lock call could potentially fail or return an aliased pointer directly into video memory, resulting in terrible performance.

When using this flag to create a mesh, we have to be very careful because many of the mesh functions (GenerateAdjacency, OptimizeInPlace, and even D3DXLoadMeshFromX) require read access to the vertex and index buffers. Specifying this flag will cause these functions to fail. Note that even though this flag can be specified in the D3DXLoadMeshFromX function, the function will fail if this is the case.

So it would seem as if there is no way to create a mesh with the `D3DXMESH_VB_WRITEONLY` flag and then optimize it, because to optimize a mesh we first need to generate the adjacency information (which will fail if the vertex buffer is write-only). Furthermore, the vertex buffer itself will usually be read from and reordered during an attribute sort (which will also fail if the mesh has a write-only vertex buffer).

To get the benefit of optimized write-only vertex buffers in our meshes, we will load the mesh into a temporary mesh (usually in system memory) created without the `D3DXMESH_VB_WRITEONLY` flag. At this point, adjacency information can be generated and optimization performed. Finally, we will clone the mesh and specify that the clone have the write-only flag set. We will discuss mesh cloning later in this chapter.

D3DXMESH_VB_DYNAMIC
This flag is analogous to specifying `D3DUSAGE_DYNAMIC` when creating a normal vertex buffer. Because dynamic vertex buffers carry some overhead due to a buffer swapping mechanism that allows the buffer to be locked without stalling the pipeline, you should only create a mesh with a dynamic vertex buffer if you intend to lock the and alter the vertex buffer contents in time critical situations. Dynamic vertex buffers were discussed in Chapter Three.

D3DXMESH_VB_SOFTWAREPROCESSING
This flag is analogous to the `D3DUSAGE_SOFTWAREPROCESSING` flag that we can specify when creating a vertex buffer manually. When using a mixed-mode device, we must indicate when a vertex buffer (and therefore a mesh) is created whether we intend to render it when the device is in software or hardware vertex processing mode. By default, the mesh vertex buffer will be created for hardware vertex processing. If we intend to use the mesh in software vertex processing mode on a mixed-mode device, we must specify this flag.

D3DXMESH_IB_SYSTEMMEM
D3DXMESH_IB_MANAGED
D3DXMESH_IB_WRITEONLY
D3DXMESH_IB_DYNAMIC
D3DXMESH_IB_SOFTWAREPROCESSING
The five flags listed above control how the index buffer of the mesh is generated. They are analogous to their vertex buffer counterparts previously discussed. Vertex buffers and index buffers are stored on the same type of memory surface, so the semantics of both are the same.

D3DXMESH_VB_SHARE
Later in the chapter we will discuss mesh cloning using the ID3DXBaseMesh::Clone and ID3DXBaseMesh::CloneFVF functions. By default these functions create a new mesh object and copy the data from the original mesh into its vertex, index, and attribute buffers. When we specify this flag as an option to the ID3DXBaseMesh::Clone function, the result will be a new mesh object that shares the vertex buffer with the original mesh. This is often referred to as *instancing* (see Chapter One). Any modifications made to the master vertex buffer will affect all cloned meshes. It should be noted that this is not a valid mesh creation flag for the D3DXCreateMesh or D3DXLoadMeshFromX functions. It should only be used in a mesh cloning operation.

D3DXMESH_USEHWONLY
This flag is only valid for the ID3DXSkinInfo::ConvertToBlendedMesh function. It creates a mesh that has per-vertex blend weights and a bone combination table from a standard ID3DXMesh. Skinned meshes will be covered later in this course, so we can ignore this flag for now.

The next five members are combinations of the previous flags discussed. They emerge from the concept that we will often create both the index buffer and the vertex buffer in the same resource pools using the same functionality.

D3DXMESH_SYSTEMMEM
Informs the mesh creation functions that we wish both the vertex buffer and the index buffer of the mesh to be created in system memory. Equivalent to specifying both `D3DXMESH_VB_SYSTEMMEM` and `D3DXMESH_IB_SYSTEMMEM`.

D3DXMESH_MANAGED
Informs the mesh creation functions that we wish both the vertex buffer and the index buffer of the mesh to be created in the managed memory pool. Equivalent to specifying both `D3DXMESH_VB_MANAGED` and `D3DXMESH_IB_MANAGED`.

D3DXMESH_WRITEONLY
Specifying this flag will create the mesh index buffer and vertex buffer with the write-only flag. Equivalent to specifying both `D3DXMESH_VB_WRITEONLY` and `D3DXMESH_IB_WRITEONLY`.

D3DXMESH_DYNAMIC
Equivalent to specifying both `D3DXMESH_VB_DYNAMIC` and `D3DXMESH_IB_DYNAMIC`. This will create a mesh that can have both its vertex buffer and index buffer efficiently locked without stalling the pipeline.

D3DXMESH_SOFTWAREPROCESSING
Equivalent to specifying both `D3DXMESH_VB_SOFTWAREPROCESSING` and `D3DXMESH_IB_SOFTWAREPROCESSING`. Used to specify when using a mixed mode device that the mesh will be rendered in software vertex processing mode.

We have not covered all of the flags that can be specified as the second parameter to the D3DXLoadMeshFromX function. However you will usually find yourself using only one or two of the flags we have looked at here. Please consult the SDK documentation if you would like more information on other flag types.

LPDIRECT3DDEVICE9 pDevice
This is a pointer to the device interface. Mesh objects are always owned by the device object because the vertex and index buffers themselves are owned by the device object.

LPD3DXBUFFER * ppAdjacency
This is the address of an ID3DXBuffer pointer. It should *not* point to an already valid buffer interface since the function creates the buffer object when called. If the function is successful and the mesh is successfully created, this pointer will point to a valid ID3DXBuffer interface that contains the face adjacency information for the mesh. There will be three DWORDs in this buffer for every triangle in the mesh. If your application does not require the face adjacency information, you can simply release this buffer.

LPD3DXBUFFER *ppMaterials
This buffer represents an array of D3DXMATERIAL structures. The D3DXMATERIAL structure is a superset of the D3DMATERIAL9 structure and contains both a D3DMATERIAL9 and a texture filename.

```
typedef struct D3DXMATERIAL
{
      D3DMATERIAL9 MatD3D;
      LPSTR pTextureFilename;
} D3DXMATERIAL;
```

There is one structure in the array for each mesh subset. Thus, if the numbers in the attribute buffer range from 0 to 5 (indicating six subsets) then this buffer will store six D3DXMATERIAL structures. The subset ID itself is the index of the D3DXMATERIAL in the array.

The D3DXMATERIAL structure describes a texture/material pair. This information is stored in the X file and describes the texture and material information for each subset. For example, before calling DrawSubset and passing a value of 0 for the first subset of the mesh, we first set the material and texture described in the first element of the D3DXMATERIAL array.

Note that this information will not be maintained by the mesh object after the call. The application must store the material and texture information in its own arrays for correct subset rendering. Also note that while the material information is stored in the X file, only the texture filename is returned. It is the application's responsibility to load the texture.

While the term ‘material’ is actually quite commonly used in 3D graphics programming circles to describe all of the rendering attributes for a surface, it can be somewhat confusing given the limited definition of a material that we have been using since studying lighting in Chapter Five. The D3DXMATERIAL structure might have been more appropriately named something like D3DXATTRIBUTE or D3DXSUBSETATTRIBUTE to more accurately describe what each element in the material buffer holds, and to minimize confusion. As it stands, just try to keep this in mind moving forward. We can see that because each structure holds the texture and material used by a given subset, two separate subsets that have unique material information but share the same texture information will be represented by two unique D3DXMATERIAL structures in the buffer. Therefore, as multiple D3DXMATERIAL structures may reference the same texture, we must make sure that we do not load the texture multiple times.

Something else which is worth noting is that the X file format supports vertices with multiple sets of texture coordinates but supports only a single texture per face. That is, there is no explicit multi-texture support in the default X file format (although you can create custom chunks to support this feature).

Another small note about the X file format is that it does not store a material ambient property. Although the D3DXMATERIAL structure encapsulates a D3DMATERIAL9 structure (which does indeed support an ambient color), it will always be set to 0 for all materials extracted from the X file. This can create some degree of confusion if you are not aware of this limitation, so it might be wise to set the ambient color of the material to (1,1,1,1) when copying the material data out of the buffer. This will allow for controlling the ambient setting with the `D3DRS_AMBIENT` render state.

DWORD *pNumMaterials
This is the address of a DWORD which will contain the number of materials in the D3DXMATERIAL buffer. Consequently, this value also describes the number of subsets in the mesh and informs us that the attribute buffer will contain Attribute IDs in the range [0, NumMaterials-1].

LPD3DXBUFFER* ppEffectInstances
In Module III we will study Effects, which are essentially a high level abstraction that encapsulates rendering concepts like texture states, transforms states, render states, and more. This buffer is an array of D3DXEFFECTINSTANCE structures (one for each subset). Each structure contains the filename of the effect file that should be used for rendering the subset and an array of default values that can be passed into the effect file code. This allows an artist or modeler to store the default parameters for an effect in the X file. It should be noted that the X file does not store the actual effect file. Effect files will need to be loaded by the application as was the case with textures stored in the material buffer.

LPD3DXMESH* ppMesh
The final parameter is the address of a pointer to an ID3DXMesh interface. If the function is successful this will point to a valid mesh interface when the function returns.

The following code snippet demonstrates how we might use this function to load an X file into an ID3DXMesh. In this example it is assumed that the CScene class has an array allocated to store D3DMATERIAL9 structures and an array of TEXTURE_ITEM structures to store the textures and their filenames. We use a structure rather than just storing the texture interface pointers so that we can store the texture filename to make sure we do not load it more than once.

```
typedef struct _TEXTURE_ITEM
{
    LPSTR               FileName;   // File used to create the texture
    LPDIRECT3DTEXTURE9  Texture;    // The texture pointer itself.
} TEXTURE_ITEM;
```

The following code loads the X file and immediately releases the returned adjacency and effect instance buffers because they are not needed in this example. After that, we get a pointer to the returned D3DXMATERIAL buffer and start copying the material information to our CScene class D3DMATERIAL9 array and load the textures and into the CScene::m_pTextureList array.

```
ID3DXBuffer       *pAdjacency;
ID3DXBuffer       *pSubSetAttributes;
ID3DXBuffer       *pEffectInstances;
ID3DXMesh         *pMesh;
DWORD              NumAttributes;

D3DXLoadMeshFromX("Cube.x", D3DXMESH_MANAGED, pDevice , &pAdjacency,
                  &pSubSetAttributes, &pEffectInstances, &NumAttributes, &pMesh);

// We don't need this info in this example so release it
pAdjacency->Release();
pEffectInstances->Release();

// Lets get a pointer to the data in the material buffer
D3DXMATERIAL *pXfileMats = (D3DXMATERIAL*) pSubSetAttributes->GetBufferPointer();
```

At this point we can use the pointer to the D3DXMATERIAL data to step through the material buffer and process the information. The first step copies the D3DMATERIAL9 information into the CScene material array. Since an X file does not contain ambient information, we set the ambient property to opaque white after the copy.

```
// Loop through and store all the materials and load and store all the textures
for ( DWORD  i = 0 ; i < NumAttributes ; i++ )
{
    // first copy the material data into the scenes material array
    pScene->m_Materials[i] = pXFileMats[i].MatD3D;
    pScene->m_Materials[i].Ambient = D3DXCOLOR ( 1.0f , 1.0f , 1.0f , 1.0f );
```

Now we will process the texture for the current attribute. We will check to see whether any of the previous textures we have loaded share the same filename and if so, store a copy of the pointer in the current TEXTURE_ITEM element and increase the texture's reference count.

```
    // Used to track the whether a texture has already been loaded.
    BOOL bTextureFound = FALSE;

   // Next we need to check if the texture filename already exists in our
   // scene. If so, we don't want to load it again. We will just store its
   // pointer in the new array slot.

   for ( ULONG q = 0; q < pScene->m_TextureCount; ++q)
   {
```

```
        if( _tcsicmp( pScene->m_pTextureList[q]->FileName,
                  pXFileMats[i].pTextureFilename ) == 0 )
        {
                pScene->m_pTextureList[i]->Texture=
                                        pScene->m_pTextureList[q]->Texture;

                pScene->m_pTextureList[i]->Filename=
                           strdup(pScene->m_pTextureList[q]->Filename);

                pScene->m_pTextureList[i]->AddRef();

                bTextureFound = TRUE:
        }

    } // Next Texture
```

If the loop ends and bTextureFound is still false, then this is a new texture and must be loaded and the filename stored for future comparisons.

```
     // If the texture did not already exist load it into our array
     if( !bTextureFound )
     {
        D3DXCreateTextureFromFile(pScene->m_pD3DDevice,
                                  pXFileMats[i].pTextureFilename,
                                  &pScene->m_pTextureList[i]->Texture);

        pScene->m_pFilename = strdup(  pXFileMats[i].pTextureFilename );
     }


     // a new texture has been added to our CScene TEXTURE_ITEM array.
     pScene->m_TextureCount++;

} // next material in buffer
```

The code above requires some tighter error checking (for example, a material may not have a valid texture filename) but it gives us a good idea of the basic steps.

At this point, each face in our mesh has an Attribute ID assigned and stored in the attribute buffer. This ID describes the index of the material and texture it should be rendered with. With this in mind, we can render our mesh like so:

```
pDevice->SetFVF( pMesh->GetFVF() );

// render our mesh
for ( DWORD i = 0; i < NumAttributes; i++ )
{
      pDevice->SetMaterial ( &pScene->m_Materials[i] );
      pDevice->SetTexture  ( 0 , pScene->m_pTextureList[i].Texture );
      pMesh->DrawSubset    ( i );
}
```

8.6.2 D3DXLoadMeshFromXInMemory

We can also load the X file data in its entirety into a block of memory and instantiate a mesh object using this stored data. The block of memory should be an exact representation of the X file (headers, etc.). This can be done by inquiring about the size of the file, allocating a block of memory to accommodate it, and then reading every byte of the file into the memory buffer. Once the file is in memory, you can use the following function to create an ID3DXMesh:

```
HRESULT D3DXLoadMeshFromXInMemory
(
    LPCVOID Memory,
    DWORD SizeOfMemory,
    DWORD Options,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXBUFFER *ppAdjacency,
    LPD3DXBUFFER* ppMaterials,
    LPD3DXBUFFER* ppEffectInstances,
    DWORD *pNumMaterials,
    LPD3DXMESH *ppMesh
);
```

This function is analogous to the texture loading function D3DXCreateTextureFromFileInMemory (see Chapter Six) in that rather than loading the X file from a file on the hard disk, it loads it from data that is stored in memory. With the exception of the first two parameters, all other parameters are identical to the D3DXLoadMeshFromX function.

LPCVOID Memory
This is a pointer to the first byte in the block of memory containing the X file data that has been loaded.

DWORD SizeOfMemory
This parameter describes the size of the X file data memory block in bytes.

8.6.3 D3DXLoadMeshFromXResource

Sometimes storing models as resources -- which are compiled either inside the application module or an external compiled module -- is preferred since it prevents users from gaining access to your models outside of the application. For smaller applications this is fine, but we will look at better alternatives for keeping our data safe in Module III.

The last seven parameters are identical to those described in the D3DXLoadMeshFromX function. The first three parameters are used to pass information about the module the X file resource is stored in, the name of the resource within the module, and the type of resource that it is.

```
HRESULT D3DXLoadMeshFromXResource
(
    HMODULE Module,
    LPCTSTR Name,
    LPCTSTR Type,
    DWORD Options,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXBUFFER *ppAdjacency,
    LPD3DXBUFFER *ppMaterials,
    LPD3DXBUFFER* ppEffectInstances,
    DWORD *pNumMaterials,
    LPD3DXMESH *ppMesh
);
```

HMODULE Module
This is the handle of an already loaded module that contains the resource.

LPCTSTR Name
This is the name that has been given to the X file resource in the module. It is used to extract the correct resource from the specified module.

LPCTSTR Type
This is a string describing the resource type.

8.6.4 D3DXLoadMeshFromXof

```
HRESULT D3DXLoadMeshFromXof
(
    LPDIRECTXFILEDATA pXofObjMesh,
    DWORD Options,
    LPDIRECT3DDEVICE9 pD3DDevice,
    LPD3DXBUFFER *ppAdjacency,
    LPD3DXBUFFER *ppMaterials,
    LPD3DXBUFFER* ppEffectInstances,
    WORD* pNumMaterials,
    LPD3DXMESH *ppMesh
);
```

This function is used to work with lower level X file parsing routines. While DirectX Graphics now provides most of the high level X file loading and creation functions that we should need, many of these functions were not available in earlier versions of DirectX. Even DirectX 8 had no mechanism to automatically load mesh hierarchies (Chapter Nine), so X files had to be loaded at a lower level. Even now, it is possible that you may still need to resort to lower level X file loading -- if you intend to place customized template types in your X files, then this is certainly the case.

From the very early days of DirectX, a set of interfaces has been available to parse X files. Although a complete discussion of these low level interfaces is beyond the scope of this chapter, there is some

coverage included in the next chapter and in the DX9 SDK documentation itself (check the section called 'The X File Reference'). For now, we will briefly explain the basics of how these interfaces work.

The first thing we need to do to load data manually from an X file is call the function DirectXFileCreate. This will create a DirectXFile object and return a pointer to an IDirectXFile interface.

```
IDirectXFile *pXfile;
DirectXFileCreate ( &pXfile );
```

The IDirectXFile interface exposes three methods:

- CreateEnumObject - used for enumerating the memory chunks in the X file (meshes, materials matrices, etc.)

- CreateSaveObject - creates an interface that allows you to save data out to an X file (with custom templates if desired) \

- RegisterTemplates - used to register your own custom templates so that they are recognized by both the enumerator and save objects.

Normally the first thing we would do after creating the interface is register the standard X file templates. The following function call makes sure that the X file interfaces understand all of the common templates found in an X file.

```
pXfile->RegisterTemplates((VOID*) D3DRM_XTEMPLATES, D3DRM_XTEMPLATE_BYTES);
```

We now call the CreateEnumObject function to create an enumeration object for a given file. We will get back an IDirectXFileEnumObject interface, which we can use to step through the file contents.

```
IDirectXFileEnumObject *pEnumObject;
pXfile->CreateEnumObject( "MyXfile.x" , DXFILELOAD_FROMFILE ,&pEnumObject );
```

The first parameter is actually dependant on the second parameter. In this case, because we are specifying `DXFILELOAD_FROMFILE`, the first parameter should be the X file name. The third parameter is the address of an IDirectXFileEnumObject interface that will point to a valid enumeration object on function return.

Once we have the enumeration object, we can use it to step through the *top-level* data chunks in the X file. A top-level object in an X file is an object which has no parent, but may or may not have multiple child objects. (The concept of parent/child hierarchies will be discussed in detail in the next chapter).

We use the IDirectXFileEnumObject::GetNextDataObject function to fetch the next top-level data object from the X file. It will return a pointer to an IDirectXFileData interface representing the retrieved data object. This interface can then be used to inquire about the object's properties. The following code shows how we can retrieve top-level objects in the mesh file using a while loop.

```
while (SUCCEEDED(pEnumObject->GetNextDataObject(&pFileData)))
{
    pFileData->GetType(&pGuid);

    if (*pGuid==TID_D3DRMMesh) ParseMesh ( pFileData );

    pFileData->Release();
}
```

Once we have the data object, we can call the IDirectXFileData::GetType function to return the GUID for the object type. The DirectX SDK provides a list of all object types and their associated GUIDs so that we can use them to make decisions about how and what to parse. In the simple example above, we are ignoring all top-level objects except meshes. Once we locate a mesh, we can extract the data from it using the D3DXLoadMeshFromXof as shown below.

> **Note**: If we load a mesh hierarchy manually using this approach, then we will also want to also process the TIF_D3DTMFrame top-level object and any child objects. Frame objects can contain additional child meshes and matrices describing parent-child spatial relationships. Processing a frame object is a recursive concept where we test for children and create child meshes as needed. These are stored along with their respective matrices. In our example above, child meshes and frames would be ignored and only top-level meshes would be created. In the next chapter, we will discuss scene hierarchies.

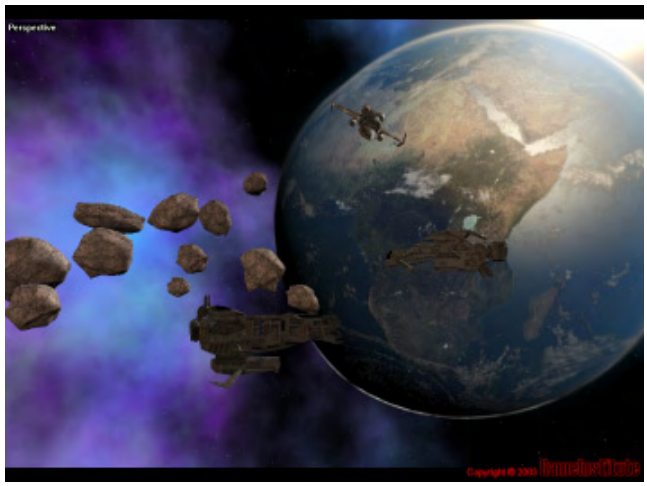
```
void ParseMesh ( IDirectXFileData * pFileData )
{
    ID3DXBuffer * pAdjacency;
    ID3DXBuffer * pMaterials;
    ID3DXBuffer * pEffects;

    D3DXLoadMeshFromXof ( pFileData , D3DXMESH_MANAGED , pDevice ,
                          pAdjacency, pglbMaterials[glbMeshCount],pEffects,
                         &pglbNumMaterials[gblMeshCount],
                         &pMeshArray[glbMeshCount] );

    glbMeshCount++;
    pAdjacency->Release();
    pEffects->Release();

}
```

The above function assumes that pglbMaterials is a global array of ID3DXBuffer interface pointers. For each mesh extracted, we can store its buffer in an array for later processing. The number of materials is stored in a global and this tells us how many materials are in the corresponding materials buffer. The returned ID3DXMesh interface is also stored in a global array. The adjacency and effect buffers are simply released as we are not using them in this example.

> **Note:** D3DXLoadMeshFromX, D3DXLoadMeshFromXInMemory, D3DXLoadMeshFromXof and D3DXLoadMeshFromResource do not support the loading of multiple meshes and arranging them in a hierarchical fashion. It is often the case (as we will see in the next chapter) that an X file may contain a number of mesh objects which may or may not be arranged hierarchically. When an X file is loaded using one of these functions, a single mesh is returned. If the file contained multiple meshes, they will be collapsed into a single mesh. We still get the full geometry set, but we lose the ability to move each sub-mesh independently. If the file contains a mesh hierarchy, such that a mesh can have child meshes and transformation matrices, this function will transform the child meshes using the matrices in the X file

before collapsing it into the returned mesh. This ensures that meshes that are defined relative to parent meshes in the X file correctly collapse into a single mesh with one coordinate space. In Chapter Nine we will examine how to load mesh hierarchies.

8.7 Mesh Creation



Often we will need to create mesh objects manually for the purpose of filling them with data from another file format (IWF for example). In Lab Project 8.1, we will create a simple mesh viewing application that supports X and IWF file loading. Some of our scene meshes will be loaded from X files using the D3DXLoadMeshFromX function and others will be created manually and populated with mesh data from an IWF file.

Creating an empty mesh is done with a call to either the D3DXCreateMeshFVF or D3DXCreateMesh functions.

D3DXCreateMesh allows us to specify a mesh vertex format using something called a *declarator*. Vertex declarators and the D3DXCreateMesh function will be discussed in Module III of this series when we examine vertex shaders. For the rest of this course, we will use the D3DXCreateMeshFVF version of the function. D3DXCreateMeshFVF allows us to describe the desired vertex format using the more familiar flexible vertex format flags.

```
HRESULT D3DXCreateMeshFVF
(
    DWORD NumFaces,
    DWORD NumVertices,
    DWORD Options,
    DWORD FVF,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXMESH *ppMesh
);
```

DWORD NumFaces
This parameter tells the function how many triangles the mesh will have. This value will be used to allocate the mesh index buffer and its attribute buffer to ensure enough space to accommodate this many triangles.

DWORD NumVertices
This parameter tells the function how many vertices will be stored in the vertex buffer. The function uses this value to allocate a vertex buffer large enough to hold only this number of vertices.

DWORD Options

The options flag has the same meaning we saw in the D3DXLoadMesh… functions. It informs the function about memory resource pools, static vs. dynamic properties and/or write-only status. Unlike D3DXLoadMeshFromX and its sister functions, you can use the `D3DXMESH_VB_WRITEONLY`, `D3DXMESH_IB_WRITENONLY` and `D3DXMESH_WRITEONLY` flags and the function will not fail. As mentioned previously however, if you do specify any of these flags, then any subsequent calls to optimize the mesh or generate face adjacency information will fail. Thus, only use the write-only flags if you do not plan on touching the buffers that have been created.

DWORD FVF

Since we are creating an empty mesh, we can choose the vertex format we wish the mesh to use. This flag will be used in conjunction with the NumVertices parameter to allocate the appropriate amount of vertex buffer memory. These FVF flags are no different than the FVF flags we have been using throughout this series when creating a vertex buffer.

LPDIRECT3DDEVICE9 pDevice

Because the device object will own the vertex and index buffer memory, we must pass in a pointer to the device interface for mesh creation.

LPD3DXMESH *ppMesh

The final parameter is the address of an ID3DXMesh interface pointer. On successful return, it will point to the newly created mesh object interface.

The following code demonstrates creating an empty mesh object with enough space in its vertex buffer for 24 untransformed and unlit vertices. The index buffer is large enough to contain 12 triangles (12*3 = 36 indices) and an attribute buffer large enough to hold 12 DWORDs -- one for each face.

```
ID3DXMesh *pMyMesh;
DWORD fvfFlags = D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1
D3DXCreateMeshFVF (24, 12, D3DXMESH_MANAGED, fvfFlags, pDevice, &pMesh);
```

At this point we have an empty mesh and we can begin filling it with our required information. This can be done by locking the buffers and copying the vertex, index, and attribute information into them. Check the source code and workbook for Lab Project 8.1 to see how this was done using IWF data exported from GILES™.

8.8 Mesh Cloning

Cloning is basically the process of duplicating mesh data. However, since we can also specify a desired vertex format, cloning is actually a bit more sophisticated than a simple copy operation. Every mesh type that is derived from ID3DXBaseMesh inherits the Clone and CloneFVF functions. Whenever we call a clone function, a new mesh will be generated and the original mesh data will be copied into the new mesh (although vertex buffer sharing is supported as well). The old mesh remains intact and can continue to be used normally.

Before we discuss some important uses for mesh cloning, let us first take a look at the ID3DXBaseMesh::CloneFVF function to see how we can perform a cloning operation. It should be noted that the ID3DXBaseMesh::Clone function (without the FVF) works with declarators rather than FVF flags. We will discuss vertex declarators in Module III of this series.

```
HRESULT ID3DXBaseMesh::CloneMeshFVF
(
    DWORD Options,
    DWORD FVF,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXMESH *ppCloneMesh
);
```

DWORD Options

These flags will be a combination or zero or more members from the D3DXMESH enumerated type that will affect the way the cloned mesh is created. We discussed the various members of this enumerated type when we examined D3DXLoadMeshFromX and their meaning is the same. This means that the cloned mesh need not have its vertex buffers allocated in the same resource pools as the original mesh. It may also have dynamic buffers where the original did not. For example, we could create a mesh in the system memory pool, optimize it, and then clone it out to a mesh in the default pool with the `D3DXMESH_WRITEONLY` flag. This would create a video memory based optimized mesh for fast rendering. This is in fact a very common use of cloning.

One member of the D3DXMESH enumerated type is only used when cloning: `D3DXMESH_VB_SHARE`. With this flag, the cloned mesh will share the vertex buffer with the original mesh and the result is a memory savings. This can be very useful for proprietary LOD algorithms or even just simple mesh instancing. Modifying the original vertex buffer will affect all meshes cloned in this way. The default behavior is to create a cloned mesh with its own vertex, index, and attribute buffers.

DWORD FVF

This value allows us to pass FVF flags for the cloned mesh vertices. The result is that we can clone a mesh into a different vertex format. This can be very useful when we load X file meshes and have no direct control over the FVF format stored in the file. By cloning the mesh into a copy that uses a new vertex format, we can create additional room for texture coordinates or vertex normals that may not have been stored in the original X file. Once we have a cloned copy of the mesh, we could release the original mesh and make the clone the final mesh that we use for rendering.

LPDIRECT3DDEVICE9 pDevice
A pointer to the device that will own the mesh resources.

LPD3DXMESH *ppCloneMesh
Provided the function is successful, this will point to a valid interface for the newly cloned mesh.

Cloning Advantages

1. *Removal of unnecessary or extraneous vertex components.* It is easy to strip off vertex components that we do not intend to support in our engine. After loading a mesh (from a file for example) we simply specify the desired FVF and clone it. The new mesh will have only the required data and we can safely release the original mesh. As an example, imagine we load a mesh from an X file which contains vertex normals, but our engine has no use for them (perhaps it is not using the lighting pipeline). We could clone a new mesh which does not have vertex normals and then release the original. Our new mesh is now a streamlined version of the original X file representation containing only the data our application requires.

In the following code, pMesh is a pointer to a mesh that was created using the D3DXLoadMeshFromX function and it contains all of the components loaded from the X file. Assume that in this particular case, the mesh includes three sets of texture coordinates, two of which we do not need. We will create a clone that contains only the vertex components we are interested in.

```
LPD3DXMESH NewMesh;
pMesh->CloneMeshFVF(D3DXMESH_MANAGED, D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1,
                    m_pD3DDevice, &NewMesh);
pMesh->Release();
```

2. *Adding missing vertex components.* We can also add vertex components that may not have been present in the original mesh. For example, if the mesh did not include vertex normals, we can add them by cloning with `D3DFVF_NORMAL` included in the flag. Note that while the clone will now have space for the required vertex components, the data itself will still need to be initialized. The cloning functions do not automatically generate the actual values for these added components. They can only make space for them in the vertices so that they can be later initialized.

In the next example, if the original mesh does not contain vertex normals, then we clone a version that does. Once we have a clone that includes space for vertex normals, we call D3DXComputeNormals (another very useful global D3DX function) to compute the vertex normals for the clone. The code assumes that m_pMesh is a pointer to a mesh that was loaded using D3DXLoadMeshFromX.

```
if(!m_pMesh->GetFVF() & D3DFVF_NORMAL)
{
    LPD3DXMESH NewMesh;

    // Introduce a normal component into our VB
    m_pMesh->CloneMeshFVF(m_pMesh->GetOptions(),
                          m_pMesh->GetFVF() | D3DFVF_NORMAL, m_pD3DDevice, &NewMesh);

    // Generate some new vertex normals
    D3DXComputeNormals( NewMesh, NULL );
```

```
    // Release old mesh and store
    m_pMesh->Release();
    m_pMesh = NewMesh;

} // End if no normals
```

3. *Building a render-ready mesh.* In certain situations, mesh cloning becomes a requirement. In a few cases, such as when working with the utility classes (ID3DXSPMesh/ID3DXPatchMesh), the clone functions exposed by these interfaces are the only way to get the data back out to a render-ready format -- an ID3DXMesh.

4. *Maximum benefit from optimization functions*. Write-only buffers make rendering more efficient, but in order for mesh optimization functions to succeed, they need read access to the buffer data. Write-only vertex buffers will cause these routines to fail. In order to achieve both ends, a common practice is to load the mesh into system memory and perform the geometry optimizations. Since the optimization process is performed by the host CPU and not the graphics card, they will be performed more rapidly on a system memory mesh anyway. Once the mesh has been optimized, we will clone the system memory mesh to a default pool or managed pool mesh. This is where we can specify the write-only flag for the clone's vertex buffer. At this point we can either release the system memory copy or store it for later use (perhaps to re-clone on device loss if using the default pool for the clone).

The following code loads a mesh into system memory and performs the optimization step. The optimized mesh is then cloned out to the default pool with the write-only flag set for both buffers.

```
ID3DXBuffer *pAdj ;
ID3DXBuffer *pMat;
ID3DXBuffer *pEff;
ID3DXMesh *pSysCopyMesh;
ID3DXMesh *pRenderCopyMesh;

// Load the X file into a system memory mesh
D3DXLoadMeshFromX("Cube.x" , D3DXMESH_SYSTEMMEM , pDevice , &pAdj , &pMat,
                  &pEff , &NumMaterials , &pSysCopyMesh );

// Release the effects buffer, we will not use it in this example
pEff->Release();
pEff = NULL;

DWORD *pAdjacency = pAdj->GetBufferPointer();

// Optimize the mesh and pass in the adjacency array
pSysCopyMesh->OptimizeInPlace(D3DXMESHOPT_VERTEXCACHE, pAdjacency, NULL, NULL, NULL);

// Optimization has been done se we no longer need the adjacency info
pAdj->Release();
pAdj = NULL;
pAdjacency = NULL;

// Clone the mesh out to a default pool mesh with write only vertex and index buffers
pSysCopyMesh->CloneFVF(D3DXMESH_WRITEONLY, pSysCopyMesh->GetFVF(),
                       pDevice, &pRenderCopyMesh);

// Release the system memory copy
pSysCopyMesh->Release();
```

To conclude our discussion of the ID3DXMesh interface, there is one more function inherited by all mesh types from the ID3DXBaseMesh interface which we should mention. It is called UpdateSemantics and it is loosely related to the Clone function in that it allows you to alter the format of a mesh vertex buffer without cloning it. However there is a limitation imposed that does not allow us to change to a format that would cause the vertex buffer to change in size. So you might use this function to change the vertex format such that a mesh uses a vertex with a specular color instead of a diffuse color. In that case we are simply changing the meaning of the DWORD stored in each vertex (a semantic difference).

> **Note:** ID3DXBaseMesh::UpdateSemantics function only works with vertex declarators (not FVF flags) so we will examine it in Module III of this series.

8.9 Progressive Meshes

In any given scene, it is probably fair to say that a number of objects will be situated at a distance from the viewer such that their full detail is no longer appreciable. Consider a model of a car that is constructed using 3000 polygons. If the car is far off in the distance, it might only take up a handful of screen space pixels when rendered. In that case the user will no longer be able to clearly see that the car has four doors, or that there is a driver sitting inside the car. Logos or other forms of writing on the car are certainly no longer able to be clearly viewed. In such a case, rendering only a handful of polygons for the car would not noticeably degrade the image presented to the viewer, given the already reduced level of detail present in the few pixels being drawn. Transforming, lighting, and rasterizing 3000 polygons under these circumstances would be inefficient to say the least. Reducing the polygon count of the object provides a significant savings with respect to both GPU processing and bandwidth.

Ideally what we would like to do is perform this process on the fly. As objects move closer to the camera, the polygon count would increase and more model detail would be visible. As objects move further away, the polygon count would be reduced. We would also prefer this detail reduction to take place in such a way that the model does not become noticeably corrupt. That is, the selection of which polygons to remove should be based on a heuristic that understands which ones contribute more than others. Certainly we want to be careful about removing polygons that would clearly distort the overall shape of the object.

What we require is a way to progressively refine the *level of detail* (LOD) of a mesh at runtime so that detail is removed when no longer needed and re-introduced when that detail is once again required.

> **Note:** While LOD is often discussed in the context of geometric manipulation, it applies to other concepts as well where we might need less detail at one time and more at another. For example, we could consider mip-mapping to be an LOD technique for textures. Animations might also be modified to be more or less complex based on factors such as view distance.

Much research in the field of mesh LOD has taken place and there are a variety of different algorithms that have become essential tools in game development. Algorithms that are based on camera distance and sometimes even orientation are called **view dependant**. View Dependant Progressive Meshes (VDPM) is the embodiment of that technique. Other algorithms are not tied directly to the camera, but instead tessellate or simplify geometry based on other criteria. These are called **view independent** algorithms and are implemented as View Independent Progressive Meshes (VIPM).

While VIPM methods do not downgrade/upgrade a mesh based on distance to the viewer per se, that is not to suggest that they cannot be used in that way. When using VIPM methods, we typically pass in a target vertex or face count for the model. This target can certainly be calculated by taking into account camera distance. Thus VIPM is more flexible in this respect and we can progressively alter mesh LOD based on whatever heuristics we choose (model distance from the camera, the capabilities of the end user's machine, a geometry detail setting that the user may choose from a menu, etc.).

VIPM techniques can be especially useful for console development where there is typically a much smaller amount of memory available to store polygons and textures. It can also be useful to achieve a desired frame rate. If the frame rate of the application drops below a certain threshold, meshes can be

simplified to lower resolutions to help achieve a more fluid gaming experience. If the frame rate is very high, more detail can be added back. In the next section we will take a brief look at the VIPM algorithm used by the D3DXPMesh and D3DXSPMesh objects to perform mesh simplification. Please note that you do not have to be aware of how VIPM works in order to use progressive mesh support in D3DX, but many students may find what is going on under the hood interesting. This will also allow you to more intelligently use the mesh interfaces in an efficient way.

8.9.1 View Independent Progressive Meshes (VIPM)

Progressive mesh support was introduced in DirectX 8.0 as part of the D3DX library. The progressive mesh support in D3DX uses a VIPM algorithm to increase/decrease the face count or vertex count to a specified vertex or face count target through requests from the calling application.

When we first create a progressive mesh, a number of calculations are done behind the scenes to build lookup tables. These tables describe how the original mesh data we pass in should be downgraded over a series of different stages. Each stage will typically store the collapse information for two triangles (although this can vary), reducing the face count of the mesh by two. This triangle removal is set in motion by a vertex being removed (i.e. *collapsed* onto a neighboring vertex). This removes an edge from the model and in turn, the triangles that share that edge. As you can imagine, a complex mesh will have many collapse stages pre-calculated for it -- each stage performing one edge collapse, typically removing a single vertex and two triangles from the mesh.

The mesh data is not actually simplified at progressive mesh creation time, but what is stored is a series of edge collapse structures. These structures can be traversed at runtime to simplify the mesh via the requests made by the application (through the ID3DXPMesh interface). With this approach, we can downgrade the mesh one stage at a time to a very low level of detail. Because the simplification stages are pre-computed at mesh creation time and simply have to be traversed at runtime in response to a simplification request, this makes the runtime part of the simplification process more efficient.

> **Note:** This is a unidirectional simplification algorithm. It is not possible to tessellate the mesh beyond the original polygon count using VIPM. Polygon splitting is not a component of the algorithm, so the original model we start with is the highest LOD we can achieve.

When specifying a target face or vertex count, the LOD routine must be able to quickly determine which vertices to remove and which triangles are affected. As mentioned, the removal of a vertex will usually result in the removal of an edge, which in turn usually removes two triangles from the index buffer. Of course, other triangles that share that vertex may also be affected and they will need to have their indices remapped to point to a neighboring vertex instead. This is the *vertex collapsing* process.

In order to go from the highest level of detail down the desired face count, the progressive mesh will step through an ordered series of pre-calculated edge collapse structures (built at mesh creation time). Each structure stores information about which vertex must be removed next, which triangles need to be removed as a result of the collapsed edge, and which remaining triangles index the removed vertex and need to have their indices remapped to a neighboring vertex instead. This very same information also describes how the simplification can be undone. We can reverse an edge collapse (a process known as a

vertex split) to re-introduce vertices and edges that have been removed. Eventually this will bring the mesh back up to its highest LOD. Again, the highest level of detail will be the dataset that the progressive mesh was first created with. While we can remove detail from this dataset and re-introduce the detail we removed, we can never introduce more detail than was contained in the original model.

> **Note:** Progressive meshes cannot add detail to the original dataset used to create the progressive mesh. No surface subdivision is ever done to the input data set. Progressive meshes can only vary the detail of a model from its original polygon count down to a lower polygon count and back up again.

Calculating which vertex to remove each time we need to collapse an edge in the mesh can be a very time-intensive task. It is usually performed by calculating an error metric that determines how much a collapse alters the surface topology of the mesh. Since this can involve testing every vertex in the vertex buffer and finding the vertex that has the lowest error (and removing that vertex), we can rest assured that the least important vertices are removed from the mesh first.

Given the time consuming nature of this task, each collapse step will be pre-calculated when the progressive mesh is first created. In this manner, the progressive mesh knows in advance the exact order that vertices will need to be removed to achieve a certain level of detail. We will see later that when we create a progressive mesh, we can influence the error metric calculation and assign priorities to make some vertices more or less important than others. Lower priority vertices will be candidates for removal early on in the simplification chain. Because the simplification routine knows the exact order that vertices will need to be removed during runtime simplification, the mesh vertex buffer can be built using the exact order of removal. Vertices that will be removed first can be stored at the end of the vertex buffer and vertices to be removed later (or not at all) can be stored at the front of the buffer. This makes LOD changes and subsequent rendering considerably more efficient.

When a simplification step is executed at runtime by our application and a vertex is removed, the D3DX object need only decrement the number of vertices in the vertex buffer for the render call. This means that the vertices currently being used to render the mesh at its current level of detail will always be in a continuous block in the vertex buffer. This is especially important if the progressive mesh is going to perform well on a software vertex processing device. As we know from our discussions in Chapter Three, a linear block of vertices will be transformed (and possibly lit) by the software pipeline.

The same logic extends to the triangles. Since D3DX will pre-calculate the order in which triangles will be removed during edge collapses, the index buffer can also be ordered accordingly. Triangles that will be removed first will be placed at the end of the index buffer and triangles that will be removed later (or not at all) will be placed near the front of the index buffer. A single edge collapse structure will tell the D3DXMesh at render time how many triangles will be taken away. If multiplied by 3, D3DX will know how many indices to ‘remove’ from the index buffer. Again, this is really just a decrement of the total count.

Fig 8.6 depicts a single pre-calculated simplification step. We see the removal of an edge by collapsing vertex 8 onto vertex 4, which causes the blue and orange triangles to be removed from the mesh. In this example, vertex 8 was determined during mesh initialization to be the first candidate for removal because it would have the least impact on the mesh topology. Note its position at the end of the vertex buffer. The current index count is also decreased by six to account for the two triangles lost during the edge collapse. This effectively pops six indices off the end of the currently used portion of the index

buffer. Triangles that were not removed but that referenced vertex 8 are then remapped to point to vertex 4.

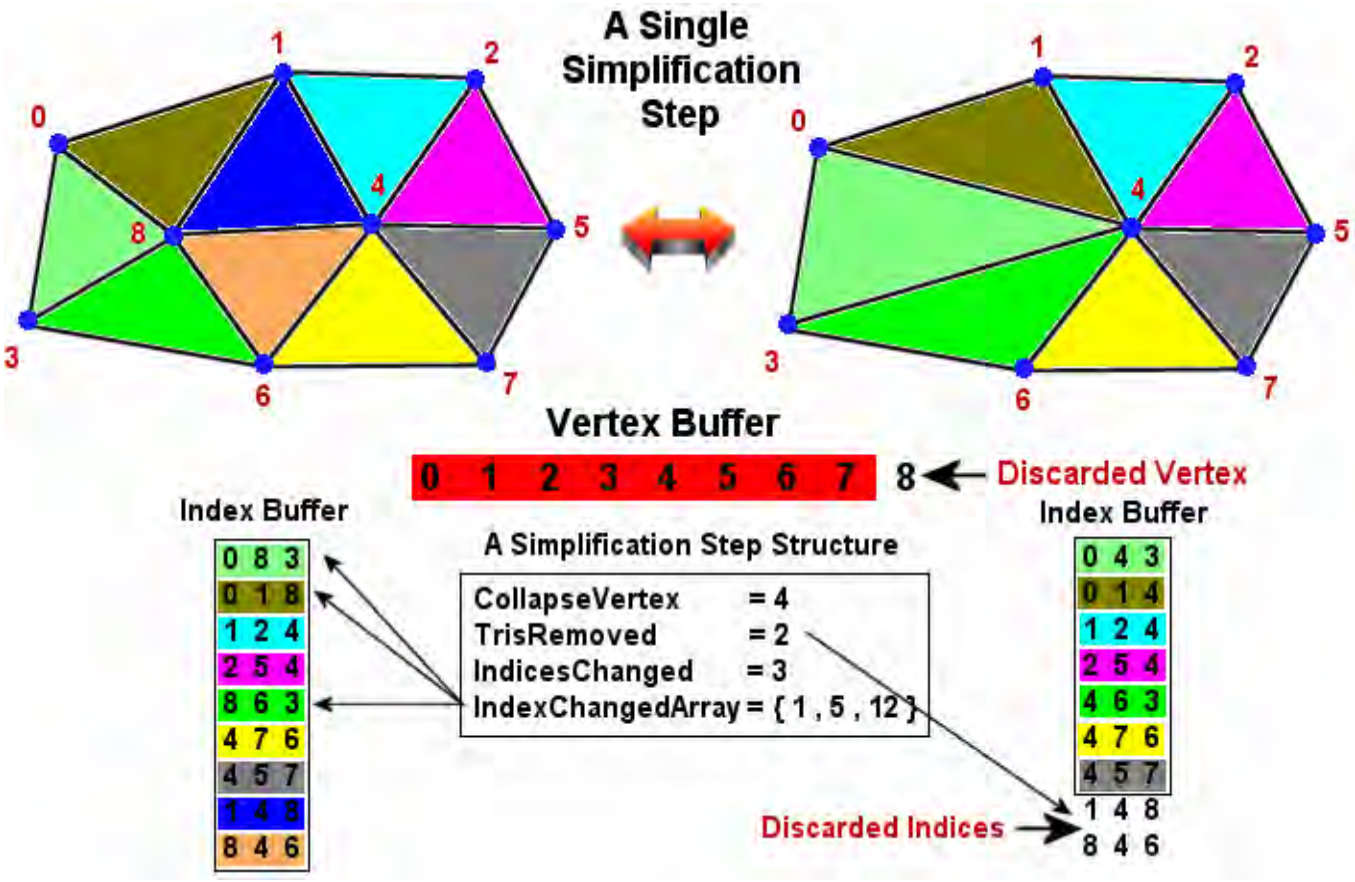


Figure 8.6

While the index remapping for the triangles that are not removed (yet still affected by the edge collapse) is done at runtime by the mesh object, each pre-computed simplification structure contains an array of indices that will need re-mapping when the simplification step is performed. In Fig 8.6 we can see that the IndexChangedArray contains the numbers of all indices in the index buffer that will need to be assigned the new vertex index because they reference the vertex which has just been removed (vertex 8). The six indices that are snipped off the end of the index buffer do not need to be remapped even if they reference the removed vertex because they are not going to be used until we undo the collapse (re-adding vertex 8 to the vertex buffer).

The following pseudo-code shows the use of the pre-compiled simplification step array (generated at mesh creation time) to simplify the mesh down to as close to the requested face count as possible. This is not necessarily the exact process carried out by the D3DX progressive mesh object mind you, but it does give us a theoretical understanding of VIPM.

```
if (CurrentFaceCount > RequestedFaceCount) // we need to process the next simplification step
{
        SimplificationStep *SimpStep = &SimplificationStepArray [CurrentStepIndex++];

        for ( I = 0 ; I < SimpStep->IndicesChanged;  I++)
        {
                pIndices[ SimpStep->IndexChangedArray[I] ] = SimpStep->CollapseVertex;
        }

        CurrentlyUsedVertices --;
        CurrentlyUsedIndices -= SimpStep->TrisRemoved * 3;
}
```

In this example the application has requested a target face count. If it is lower than the current face count then we need to process more of the pre-compiled simplification steps to further simplify the current mesh representation. If the mesh is currently set to the maximum level of detail then the first simplification step we process will be the first one in the pre-compiled array. This tells us the first vertex/edge to remove. Each simplification step tells us the indices that need to be changed to execute the current vertex removal in its IndexChangedArray. It also tells us the new values that these indices have to be set to in the structure's CollapseVertex member. Therefore, all we need to do is loop through each index in this array and change its value from the vertex about to be removed to the collapse vertex. That takes care of the triangles that will survive the collapse that reference the vertex about to be removed. Once done, we simply decrement the number of vertices we are using, effectively dropping the last vertex in the mesh, and decrease the current index count by multiplying the number of triangles dropped by three. Remember, we do not need to manually perform any of these steps, this is all hidden away inside the D3DXPMesh object. All we need to do is call a single function to set a new desired vertex or face count. The D3DXPMesh object will perform the simplification for us using a method similar to that described above.

A key point to understand is that we do not simply jump from one simplification step to another using some arbitrary position in the array. Instead we must process all of the simplification steps in between. If we currently have a face count of A and we want to simplify the mesh to face count D, we must process simplification steps A, B, C, and D to get the desired LOD. This is because a simplification step is computed relative to the one that came before it. This minimizes memory footprint for the simplification data structures.

Fig 8.7 shows that if the maximum face count of our mesh was 100, and we wanted to reduce the face count down to 90, the progressive mesh would need to carry out the first five simplification steps stored in the array. In this example we are assuming that each simplification step removes two triangles, but that may not always be the case with complex mesh topologies.

Figure 8.7

It should also be noted that this process will only be performed when the desired face count or vertex count has been modified. The resulting mesh would be used for rendering until such a time as a new target face or vertex count was specified. Thus, if we had already simplified the mesh down to 96 faces in our example, the internal simplification index would be set to 3 in Fig 8.7 because simplification steps 1 and 2 would have already been executed. If we then decided to simplify the mesh even further by another six faces, and passed in a face count of 90 (from the current 96) the simplification process would start at the current simplification level (3) and execute only simplification steps 3, 4, and 5.

Although it may not be obvious at first, the same information stored in the simplification structures we have been using to collapse edges also tell us everything we need to know to re-add that edge to the mesh and undo the simplification steps. Even at the lowest level of detail, nothing has actually been removed from the vertex or index buffers. The vertices and indices at the end of these buffers are just currently being ignored. Therefore, if we specify a desired face count that is higher than the current level of detail (but not higher than the maximum level of detail the progressive mesh was first created with), the mesh simplification routine can step backwards through the simplification array starting at the current position. Along the way it can restore the indices that were changed in the previous simplification step so that they once again point at the vertex that was removed. It can re-introduce the vertices and indices at the end of the currently unused portions of the vertex and index buffers by incrementing the counters, thereby adding the triangles that were removed in that step.

The following pseudo-code demonstrates how to reverse a simplification step and re-introduce previously deleted triangles and vertices until the desired face count is reached. It is essentially the exact opposite of the simplification code we just looked at.

```
if (CurrentFaceCount < Requested FaceCount && CurrentFaceCount < MaximumFaceCount)
{
        SimplificationStep *SimpStep = &SimplificationStepArray [ --CurrentStepIndex ];

        for ( I = 0 ; I < SimpStep->IndicesChanged; I++)
        {
                pIndices [ SimpStep->IndexChangedArray[I] ] = CurrentlyUsedVertices;
        }

        CurrentlyUsedVertices ++;
        CurrentlyUsedIndices += SimpStep->TrisRemoved * 3;
}
```

This code shows that if we need to re-introduce face detail, then we decrement the CurrentStepIndex and get a pointer to the last simplification step that was performed. The mesh has an array of indices that were changed and stored in each simplification step's IndexChangedArray. At first it might seem that we have no way of knowing which vertex those indices were re-mapped from when the collapse was performed. However, remember that for each simplification step we perform, the vertex removed is the one at the end of the currently used section of the vertex buffer. So when we are about to undo a simplification step, the next vertex just outside the currently used portion of the buffer *is* the vertex those indices were mapped to before that simplification step occurred. Therefore, we loop through each of the indices changed by that simplification step and reset their values to the CurrentlyUsedVertices number. If we currently have 10 vertices in use, then the 11^{th} vertex (i.e. Vertex[10]) will be the vertex we need to reset these indices to reference. Once we have remapped these indices to the original vertex, we simply increment the current vertex count by 1. This extends the size of the usable section of the vertex buffer to re-introduce the deleted vertex that these updated indices have now been reset to reference. Finally, we know that the triangles removed by the simplification step are still in the index buffer, but just outside the section currently being used. Therefore we increase the count of currently used indices to re-introduce the triangles that were removed.

While we do not need to know the details of VIPM and its various implementations to use the D3DX provided mesh class, it is certainly interesting material to study and may come in handy if you ever have to create your own simplification or LOD routines. This basic understanding of VIPM will also help us to properly use the D3DX supplied interface and give us some insight for decision making when it comes to issues of performance and efficiency.

> **Note:** Progressive mesh rendering, despite its potential for reduced polygon count, does not always outperform brute-force mesh rendering. This is because the algorithm requires that the vertex and index buffers be arranged in a manner such that the technique executes quickly. The buffer rearrangement will often be to the detriment of some of the optimization techniques discussed earlier (attribute sorting, vertex cache performance, etc.). Index buffer updates also hurt performance, so keep these ideas in mind as you experiment with the technique.

8.9.2 ID3DXPMesh

The ID3DXPMesh interface (derived from ID3DXBaseMesh) contains functionality that encapsulates dynamic mesh LOD. Based on our discussion in the last section, we now have a fairly good theoretical understanding of what this interface will do for us behind the scenes when we wish a mesh to be dynamically simplified.

Since its introduction in DirectX 8.0, the ID3DXPMesh has been based on the VIPM method introduced by Hughes Hoppe in a 1996 ACM SIGGRAPH paper. While the VIPM method used by the D3DX progressive mesh is not necessarily the most optimal reduction scheme, it is simple to implement, easy to use and is relatively hardware friendly. Unfortunately, it suffers from poor utilization of the vertex cache on hardware vertex processing devices given the vertex buffer ordering requirements we discussed in the last section.

It is also worth noting that the VIPM technique discussed requires the use of a dynamic index buffer to handle LOD changes. Because simplification is performed by the host CPU and involves a good deal of index buffer touching, the progressive mesh index buffer will typically be allocated in system memory. This allows the CPU to write to the buffer as quickly as possible. Although this means that we are rendering from a system memory index buffer, it is generally not as expensive as passing 2-byte indices over the bus (or 4 byte indices if 32-bit indices are used) given large sections of index data on a hardware vertex processing device. Fortunately the VIPM method used by the D3DX progressive mesh does not need to modify the vertex buffer contents once data has been properly ordered. Therefore, the vertex buffer will typically be created in video memory (local/non-local) by default on a hardware vertex processing device.

While ID3DXPMesh does introduce many additional functions beyond those inherited from ID3DXBaseMesh, most are simple ‘Get’ functions used for inquiring about the current settings or state of the mesh. Fortunately, there are surprisingly few methods that are required to use the LOD features.

8.9.3 Progressive Mesh Generation

D3DX provides no functions to load a progressive mesh from a file or even to create an empty progressive mesh. A progressive mesh is created via a standard ID3DXMesh which determines how the progressive mesh looks at its highest level of detail. D3DX exposes the D3DXGeneratePMesh function to convert a standard mesh to a progressive mesh. This provides the indirect means for creating a progressive mesh using data stored in an X file: we call D3DXLoadMeshFromX to create a normal ID3DXMesh and then feed it into the D3DXGeneratePMesh function to create a progressive mesh. Once the progressive mesh has been initialized, the ID3DXMesh interface is no longer needed and can be released. It is at this stage (when the progressive mesh is first created), that D3DX will analyze the topology of the mesh and will determine the order in which vertices and edges should be removed from the mesh. The progressive mesh will have its vertices and indices arranged in its index and vertex buffers so that triangles that will be removed first will have their vertices and indices placed at the rear of their respective buffers. After this step has been performed, the function will then pre-compute and

store all the edge collapse structures and arrange them in a list that can be traversed at runtime to perform efficient simplification in response to application requests. At this point, the function returns to the calling application a newly created progressive mesh interface ready to be simplified using a few simple ID3DXPMesh interface function calls.

```
HRESULT D3DXGeneratePMesh
(
    LPD3DXMESH pMesh,
    CONST DWORD *pAdjacency,
    CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights,
    CONST FLOAT *pVertexWeights,
    DWORD MinValue,
    DWORD Options,
    LPD3DXPMESH *ppPMesh
);
```

LPD3DXMESH pMesh

This is the input mesh from which the progressive mesh will be built. It defines the progressive mesh dataset at its highest possible level of detail. Note that this mesh is a standard ID3DXMesh which can be created manually using D3DXCreateMesh or loaded from an X file using D3DXLoadMeshFromX.

CONST DWORD *pAdjacency

Since pre-computing edge collapses relies on information describing how triangles and vertices are connected, this function requires us to pass in the adjacency information for the input mesh. This array will contain three DWORDs per-triangle and is used to compile a table of edge collapse information. The mesh will also reorganize its vertex and index buffers according to collapse order and will use this adjacency information to build a new adjacency buffer for the reorganized data. Unlike the D3DXMesh object which does not maintain adjacency information, progressive meshes will maintain their own internal face adjacency information. So once the mesh has been created, we can discard the input adjacency information of the input mesh along with the input mesh itself.

CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights

When the progressive mesh is first created, a pre-compiled simplification step lookup table will be built. It is used to order the data in the vertex and index buffers. To ensure that the mesh transitions as smoothly as possible between detail levels, each vertex receives an error value which describes how dramatic the change to the mesh topology will be if the vertex is removed. When calculating each simplification step, the vertex with the lowest error will be marked as the next to be collapsed. By default (or if we pass NULL as this parameter) the vertex position, normal, and blend weight contribute equally to the error produced for each vertex. For each vertex, a comparison is made between it and neighboring vertices to see which of the neighboring vertices is most similar to the current vertex being processed. This tells us which neighbor makes the best target for collapsing the vertex onto while keeping mesh distortion to a minimum. We would then compare various components of the two vertices and record the error value which is generated by summing the difference in the two vertex components. After doing this for each vertex, we will have a per-vertex error value describing the level of distortion that will be caused if this vertex is removed. At this point, the vertex list can be ordered such that vertices with lower error values are stored at the back of the vertex buffer and the triangles they remove pushed to the back of the index buffer.

Be default, the blend weight, vertex position, and vertex normal are used to build the error value for a vertex, and they are used in equal measure. By passing in a `D3DXATTRIBUTEWEIGHTS` structure, we can force the simplification step compiler to consider additional vertex components when generating the error for each vertex. If we pass in NULL as this parameter, a `D3DXATTRIBUTEWEIGHTS` structure with the following values is used (and is usually acceptable for most situations):

```
D3DXATTRIBUTEWEIGHTS AttributeWeights;

AttributeWeights.Position =  1.0;
AttributeWeights.Boundary =  1.0;
AttributeWeights.Normal   =  1.0;
AttributeWeights.Diffuse  =  0.0;
AttributeWeights.Specular =  0.0;
AttributeWeights.Tex[8]   =  {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
```

By default, the vertex colors and texture coordinates are not taken into account. Note that this can lead to distortion that is not topological but instead color based. If the texture coordinates or colors of a vertex that was collapsed onto another are very different, the visual results may certainly be noticeable.

This structure allows the application to change this default behavior by adding weights to additional vertex components or increase/decrease the weight of a default component such that it is considered more/less important in the comparison function and generates a larger/smaller error value, respectively. For example, the default values above show that the position, the boundary, and the normal of the vertex each contribute to the vertex error calculation equally. If we were to set the vertex normal to 2.0 instead of 1.0, we would double its importance and any error produced by differing vertex normals would be doubled. This would make vertices with different normals much less likely candidates for removal. If we were to set the normal member to 0.0, then when calculating the error of collapsing vertex A onto vertex B, the normals of vertex A and B would be ignored and would not influence the error calculation in any way.

The default values generally work pretty well most of the time and you will probably be able to pass NULL as this parameter and live with the results. However, if you discover that one of your simplified meshes is suffering severe texture distortion for example, you can weight the relevant texture coordinate set into the error calculation by using a larger value. Setting the structure's Tex[0] member to 4.0 for example, would quadruple the error caused by different texture coordinates. The result would be that two vertices with different texture coordinates are much less likely to be collapsed onto one another.

CONST FLOAT *pVertexWeights

This parameter is used to pass an array of per-vertex weights for the mesh. If NULL is passed, each vertex is assumed to have an equal weight of 1.0. Once the error for a vertex is calculated, its error is multiplied by its weight in this array, allowing the user to increase/decrease the importance of a particular vertex in a mesh.

Total Vertex Error = Vertex Error x Vertex Weight

Let us consider a mesh of a human face as an example. The nose vertices are likely to be very close together and thus may be removed from the mesh very early on in the simplification chain. Of course, the nose is a pretty important feature, so using this array we could set the weight of the nose vertices to a

larger value to ensure that they will be removed very late in the process (if at all). This allows us to retain the basic outline of the face even at a distance.

Another example might be rendering a section of terrain as a progressive mesh. As the terrain shifted to a lower LOD, a hill might disappear which currently has the mesh of a building sitting on top of it. When the hill polygons were simplified away, the building would be left floating in mid-air. Therefore, it is common practice in terrain engines that use such forms of dynamic LOD to be able to specify certain portions of the terrain mesh (such as those under buildings for example) as being non-removable.

If you do not pass NULL as this parameter, the array you pass in should hold a floating point weight value for every vertex in the input mesh. The higher the vertex weight, the less likely that vertex is to be removed. Passing NULL simply means that all vertices will be weighed equally (1.0).

DWORD MinValue

When the progressive mesh is generated, a number of simplification steps are pre-computed (one for each edge collapse). As you can imagine, a large number of edge collapse structures will be required to take a highly detailed mesh down to a handful of polygons. This value specifies a minimum number of vertices or faces (depending on the Options parameter to this function that will be discussed next) and simplification information will only be generated down to this level. This allows us to save memory and speed up the mesh creation process by instructing it not to compile and store information beyond a level of detail that we have no intention of using.

Once the mesh has been created and its simplification information pre-compiled, there is no way to simplify the mesh any lower than this number of faces or vertices. If you set the minimum number of faces to 100 for example, the edge collapse information would be generated at mesh creation time to take the mesh from its original polygon count to 100 faces and back up again. If during the render loop you called the ID3DXPMesh::SetNumFaces function and specified a value of 10, the edge collapse information for this LOD would not be available in the lookup table and the mesh would be simplified to its minimum number of faces (100 in this example).

It is worth noting that this value is not always achievable, since the mesh distortion caused by simplifying down to the requested minimum might be too extreme. However, if we do specify a minimum number that cannot be reached, the collapse information will be generated down to the lowest possible level of detail. Thus, passing zero as this parameter will always guarantee that we can simplify the mesh down to the lowest resolution considered possible by the progressive mesh object.

DWORD Options

The ***MinValue*** parameter just discussed allows us to specify the minimum requested resolution for the progressive mesh as either a minimum vertex count or a minimum face count. The ***Options*** parameter allows us to specify one member of the D3DXMESHSIMP enumerated type to tell the progressive mesh object how to interpret the ***MinValue*** parameter. The enumerated type has two members:

```
typedef enum _D3DXMESHSIMP
{
        D3DXMESHSIMP_VERTEX = 1,
        D3DXMESHSIMP_FACE = 2
} D3DXMESHSIMP;
```

LPD3DXPMESH *ppPMesh
The final parameter is the address of an ID3DXPMesh interface pointer. If the function is successful and the progressive mesh is successfully created, this will point to a valid interface for that progressive mesh object.

8.9.3.1 Data Validation and Cleaning

Certain types of geometric data may fail to be converted into a progressive mesh because the dataset is considered invalid. Common problems include twisted or invalid faces or adjacency loops that make the simplification table building process impossible. While this is generally caused by the input data being corrupt or erroneous to begin with, there is a way for us to validate whether a mesh that we have just loaded or created will make an appropriate progressive mesh. The D3DX library contains a function called D3DXValidMesh that will validate an ID3DXMesh object and return information describing whether it is suitable for progressive mesh generation or has geometric problems that will cause the progressive mesh generation process to fail.

```
HRESULT D3DXValidMesh
(
    LPD3DXMESH pMeshIn,
    CONST DWORD *pAdjacency,
    LPD3DXBUFFER *ppErrorsAndWarnings
);
```

LPD3DXMESH pMeshIn
This is the ID3DXMesh interface that we wish to test for progressive mesh compliance. If the function returns D3D_OK, then this mesh data is suitable for progressive mesh generation.

CONST DWORD *pAdjacency
This is a pointer to an array of DWORDs (3 for each triangle in the input mesh) describing the adjacency information for each triangle in the mesh. This information is generated automatically by the D3DXLoadMeshFromX function or can be requested from the input mesh using the ID3DXMesh::GenerateAdjacency function.

LPD3DXBUFFER *ppErrorsAndWarnings
If the function returns D3DERR_INVALIDMESH, this buffer will contain a *string* of errors and warnings describing where and why the mesh data contains errors. This will be useful for debugging the mesh.

If the mesh is not suitable for progressive mesh generation, the D3DX library provides a function called D3DXCleanMesh which can be used to try to repair the invalid geometry. So before we generate a progressive mesh from an ID3DXMesh, we should call the D3DXValidMesh function to test for compliance, and if the standards are not met, call D3DXCleanMesh to repair the data.

D3DXCleanMesh actually creates a new mesh along with new adjacency information. The original mesh and adjacency information buffer are not altered by the function so you can keep them if you wish. However you will generally want to release them and use the new data this function produces.

```
HRESULT D3DXCleanMesh
(
    D3DXCLEANTYPE CleanType,
    LPD3DXMESH    pMeshIn,
    const DWORD  *pAdjacencyIn,
    LPD3DXMESH   *ppMeshOut,
    DWORD        *pAdjacencyOut,
    LPD3DXBUFFER *ppErrorsAndWarnings
);
```

D3DXCLEANTYPE CleanType

This first method allows you to control how aggressively the cleanup is performed and control exactly what is considered for cleaning. It can be one of the following enumeration members. Note that although there are actually five cleaning options, there are only really two cleaning techniques you can switch on and off: the merging of back faces and front faces that share the same space, and the removal of bowties.

```
typedef enum _D3DXCLEANTYPE
 {
    D3DXCLEAN_BACKFACING = 1,
    D3DXCLEAN_BOWTIES    = 2,
    D3DXCLEAN_SKINNING   = D3DXCLEAN_BACKFACING,
    D3DXCLEAN_OPTIMIZATION   = D3DXCLEAN_BACKFACING,
    D3DXCLEAN_SIMPLIFICATION = D3DXCLEAN_BACKFACING|D3DXCLEAN_BOWTIES
} D3DXCLEANTYPE;
```

D3DXCLEAN_BACKFACING

Merge triangles that share the same vertex indices but have face normals pointing in opposite directions (back-facing triangles). Unless the triangles are not split by adding a replicated vertex, mesh adjacency data from the two triangles may conflict. This is something you will usually want to do before performing optimization on a mesh or converting that mesh to a skin (skins will be discussed later in the course).

D3DXCLEAN_BOWTIES

If a vertex is the apex of two triangle fans (a bowtie) and mesh operations will affect one of the fans, then split the shared vertex into two new vertices. Bowties can cause problems for operations such as mesh simplification that remove vertices, because removing one vertex will affect two distinct sets of triangles.

D3DXCLEAN_SKINNING

Use this flag to prevent infinite loops during skinning setup mesh operations. It is the same as specifying D3DXCLEAN_BACKFACING.

D3DXCLEAN_OPTIMIZATION

Use this flag to prevent infinite loops during mesh optimization operations. It is the same as specifying D3DXCLEAN_BACKFACING

D3DXCLEAN_SIMPLIFICATION
Use this flag to prevent infinite loops during mesh simplification operations. Performs back face merging and bowtie removal.

LPD3DXMESH pMeshIn
This is the interface pointer to the ID3DXMesh that you would like to try to repair before progressive mesh generation. It will typically be a mesh that failed the validation test. This mesh is not affected by the operation and the clean data is output in a separate mesh. After the mesh has been cleaned, this input mesh can be released.

CONST DWORD *pAdjacencyIn
This is where you pass in the pointer to the input mesh's adjacency information. This array should contain 3 DWORDs per triangle in the input mesh. This adjacency buffer can be released after the function returns and a new cleaned mesh has been generated as it may no longer describe the adjacency information of the output mesh correctly. It is merely used for the cleaning process and can be safely discarded on function return.

> **Note:** For virtually all functions and methods that expect adjacency input information, we can specify NULL for this parameter. If we do, the function will internally generate the mesh adjacency information by calling the mesh's GenerateAdjacency method behind the scenes.

LPD3DXMESH *ppMeshOut
This is the address of an ID3DXMesh pointer that will point to a new mesh containing the cleaned mesh data. If the input mesh did not require cleaning, then this interface will simply be another interface to the input mesh object. The input mesh interface can still be safely released as the reference count mechanism will prevent the underlying mesh object from being destroyed.

DWORD *pAdjacencyOut
This is a pointer to a pre-allocated buffer that the function will fill with the cleaned mesh's adjacency information. It should be large enough to store three DWORDs for each triangle in the cleaned mesh. It is worth noting that the adjacency input buffer can also be used as the adjacency output buffer (i.e. you can pass in the same pointer for both pAdjacencyIn and pAdjacencyOut). The input adjacency information will be overwritten with the new adjacency information for the cleaned mesh in that case.

LPD3DXBUFFER *ErrorsAndWarnings
This is the address of an ID3DXBuffer pointer that will be used to allocate a buffer of error information if the mesh cleaning fails.

Generating a progressive mesh in our code might look as follows:

```
HRESULT      hRet;
LPD3DXMESH   pStandardMesh;
LPD3DXPMESH  pProgressiveMesh;
ID3DXBuffer *pAdj;
ID3DXBuffer *pEff;
ID3DXBuffer *pMat;
DWORD        NumMaterials;
DWORD       *pAdjacency;
```

```
// Load our standard mesh
hRet = D3DXLoadMeshFromX("Factory.x" ,D3DXMESH_MANAGED, pDevice, &pAdj, &pMat,
                         &pEff, &NumMaterials , &pStandardMesh);

if (FAILED(hRet)) return false;

// We wont use the effects buffer so release it
pEff->Release();

pAdjacency = (DWORD*) pAdj->GetBufferPointer();

// Check to see if mesh is valid
hRet = D3DXValidMesh( pStandardMesh, pAdjacency, NULL );

if (FAILED(hRet))
{
    LPD3DXMESH pCleanedMesh;
    // Attempt to repair the mesh
    hRet = D3DXCleanMesh( pStandardMesh, pAdjacency, &pCleanedMesh, pAdjacency );


            if (FAILED(hRet))
            {
                pMat->Release();
                pStandardMesh->Release();
                pAdj->Release();
                return false;
            }

    // We repaired ok, let's store pointer
    pStandardMesh->Release();
    pStandardMesh = pCleanedMesh;

} // End if invalid mesh

// Generate our progressive mesh and request simplification down to potentially
// 10 faces
hRet = D3DXGeneratePMesh(pStandardMesh, pAdjacency, NULL, NULL, 10,
                         D3DXMESHSIMP_FACE, &pProgressiveMesh );

            if (FAILED(hRet))
            {
                pMat->Release();
                pStandardMesh->Release();
                pAdj->Release();
                return false;
            }

// We're done, release our original mesh
pStandardMesh->Release();
```

In the code above we loaded X file data into a standard ID3DXMesh and then validated it. If it passed validation then we generated a progressive mesh with the requested minimum level of detail of 10 faces.

If the mesh failed the validation, then we cleaned it, and if the mesh was cleaned successfully, went on to generate the progressive mesh from the newly cleaned mesh.

> **Note:** When a progressive mesh is first generated, its current level of detail is initially set to its lowest level of detail.

8.9.4 Setting LOD

Because ID3DXPMesh is derived from ID3DXBaseMesh, all of the same rendering functionality is available. To render the progressive mesh, we loop through each of its subsets and call ID3DXPMesh::DrawSubset in exactly the same way that we rendered a standard mesh. This call will render the mesh at its currently set level of detail.

To increase/decrease the current level of detail is quite simple: we just specify a desired vertex count or a desired face count. Given what we have learned thus far, we cannot specify a face or vertex count that is lower than the minimum value specified during progressive mesh creation because precompiled information will not be available down past that level. We also cannot specify a vertex or face count that is higher than the vertex or face count of the original model since this describes the maximum level of detail for the mesh. Specifying counts outside the allowed range will result in clamping to the minimum or maximum level of detail allowed by the mesh.

ID3DXPMesh exposes two functions to allow for current LOD configuration of either the vertex or face counts: ID3DXPMesh::SetNumFaces and ID3DXPMesh::SetNumVertices.

```
HRESULT SetNumFaces( DWORD Faces );
```

DWORD Faces
The only parameter to this function is the number of faces we would like for the current LOD. Clamping will occur if the value is outside the min/max ranges discussed previously. It should also be noted that this value is a request only. While the progressive mesh will try to achieve the requested face count, the exact face count may not be able to be reached. The final LOD may have more or fewer faces than the requested value.

```
HRESULT SetNumVertices( DWORD Vertices );
```

DWORD Vertices
This parameter tells the function the desired number of vertices for the current LOD. As with the previous function, clamping will occur if the value is outside the min/max ranges and the value is a request only.

These two functions are all we need to perform dynamic LOD on our meshes (we will usually use one or the other). In the following pseudo code, we see how we might subtract detail from the object based on whether the current frame rate has dropped below a target threshold.

```
// If the frame rate has dropped unacceptably low, reduce the mesh face count in increments of
// 10 until (hopefully) it becomes acceptable
if ( CurrentFrameRate < MinimumFrameRate )
{
        pMesh->SetNumFaces( pMesh->GetNumFaces() – 10 );
}
else
// If the current frame rate is some way over an acceptable frame rate, we can introduce more
// detail if available, making sure we create a richer environment on higher powered machines.
if ( CurrentFrameRate > AcceptableFrameRate + 10 )
{
       pMesh->SetNumFaces( pMesh->GetNumFaces() +10 );
}

// Render the mesh
for ( int I = 0; I < NumOfAttributes; I++)
{
          pMesh->DrawSubset (I);
}
```

8.9.5 Retrieving LOD

As discussed in the last section, there are min/max face and vertex counts that define the extreme LOD ranges in a progressive mesh. It is useful to know these thresholds when performing runtime simplification so that you do not waste time calling ID3DXPMesh::SetNumFaces function to add/reduce the face/vertex count of a mesh that is already at its maximum or minimum level of detail. ID3DXPMesh exposes four functions to retrieve these values.

`DWORD GetMaxFaces(VOID)`
This function returns the maximum number of faces the mesh can be set to. Any calls to ID3DXPMesh::SetNumFaces that pass a value larger than the value returned by this function will be clamped to this value. The mesh can never have more faces than the value returned by this function.

`DWORD GetMaxVertices(VOID)`
This function returns the maximum number of vertices the mesh can be set to with the ID3DXPMesh::SetNumVertices function. If we specify a value to the ID3DXPMesh::SetNumVertices function that is larger than the value returned from this function, the value will be clamped to this max vertex value.

`DWORD GetMinFaces(VOID)`
This function returns the minimum number of faces the mesh can be set to. Any calls to ID3DXPMesh::SetNumFaces that pass a value smaller than the value returned by this function will be clamped to this value. The mesh can never have fewer faces than the value returned by this function.

`DWORD GetMinVertices(VOID)`
This function returns the minimum number of vertices the mesh can be set to with the ID3DXPMesh::SetNumVertices function. If we specify a value to the ID3DXPMesh::SetNumVertices

function that is smaller than the value returned from this function, the value will be clamped to this minimum vertex value.

It is also useful to know the current LOD the progressive mesh is using, so ID3DXPMesh exposes two functions that return the current number of vertices used by the mesh and the current number of triangles. As mentioned in the last section, when we set the LOD using SetNumFaces or SetNumVertices, the exact face or vertex count requested may not actually be achievable and the mesh will be simplified as near as possible to the value requested. Therefore, after we call SetNumFaces or SetNumVertices we can use GetNumFaces and GetNumVertices to retrieve the exact face or vertex count the mesh was simplified to.

`DWORD GetNumFaces(VOID)`
Returns the current number of triangles used to render the mesh at its current level of detail.

`DWORD GetNumVertices(VOID)`
Returns the current number of vertices used to render the mesh at its current level of detail.

8.9.6 LOD Trimming

When we set the current level of detail using either SetNumVertices or SetNumFaces, we know that no vertices or triangles are physically being deleted from the vertex and index buffers. Only count values are being incremented and decremented to determine which sections of our buffers are used to render the mesh. However, at some point we may want to physically and persistently reduce the dynamic simplification range of the mesh by *trimming* the mesh. Trimming allows us to lower the maximum level of detail and increase the minimum level of detail that the mesh can be set to. Trimming is much more than simply setting a new minimum and maximum value. Instead, memory is physically freed as a result of this process.

Trimming is useful if you decide that you do not need the mesh to ever use its maximum LOD -- in which case you can set a new lower maximum. The mesh can free the vertices and indices at the end of its buffers as well as the memory taken up by the pre-compiled simplification steps that process the edge collapses at these higher LODs.

While it is clear that reducing the maximum level of detail will free up memory and result in a persistent change, it might not be immediately obvious why setting a new (higher) minimum level of detail would be beneficial. The reason is that when we specify a minimum LOD at mesh creation time, a series of edge collapse information structures have to be stored (one for each edge collapse) describing the step-by-step simplification (edge-by-edge) down to the minimum LOD. If we decide that we do not wish to simplify the mesh beyond a certain point, then we can release the memory used by the simplification steps at the bottom of the edge collapse chain. So ‘trimming’ is an appropriate description of what is occurring. We are contracting the dynamic range of the mesh and making it more memory efficient.

> **Note:** Trimming is a permanent change. Once you trim the maximum level of detail for example, you can never increase that maximum value. Likewise, once you increase the minimum level of detail, the edge

collapse information at the bottom of the simplification chain is discarded and you can never decrease that minimum value. The data is gone for good.

ID3DXPMesh exposes two trimming functions to alter the dynamic simplification range of the progressive mesh:

```
HRESULT TrimByFaces
(
    DWORD NewFacesMin, DWORD NewFacesMax,
    DWORD *rgiFaceRemap, DWORD *rgiVertRemap
);

HRESULT TrimByVertices
(
    DWORD NewVerticesMin, DWORD NewVerticessMax,
    DWORD *rgiFaceRemap,  DWORD *rgiVertRemap
);
```

The first two parameters to each function are the new dynamic range of the progressive mesh. In the case of TrimByFaces, these are the new minimum and maximum face count. NewFacesMin must be greater than or equal to the current minimum number of faces and NewFacesMax must be less than or equal to the current maximum number of faces. The same logic holds true for TrimByVertices for vertex counts instead of face counts.

Both functions accept two pointers to two pre-allocated DWORD buffers which return the vertex and face re-mapping information caused by the trimming function. The trimming function may cause vertices and indices to be shuffled about inside the index and vertex buffers and this could be problematic if you have external objects or structures linked to vertices or faces by index. Although both of these parameters can be set to NULL (and probably will be most of the time), if you need to know where vertices or faces have moved, you can supply these buffers and the function will fill them.

The rgiFaceRemap parameter should be large enough to hold one DWORD for every face in the pre-trimmed mesh's maximum LOD (its maximum face count). When the function returns, each element in the array will describe how that face has been mapped to another. If rgiFaceRemap[5] equals 10 for example, then this means the face that was originally 6^{th} in the index buffer has now been moved to the 11^{th} slot in the buffer.

The rgiVertexRemap parameter should either be set to NULL or it should point to a buffer large enough to hold one DWORD for every vertex in the pre-trimmed mesh's maximum vertex count. When the function returns, this buffer will describe the new location of a vertex in the vertex buffer. If rgiVertexRemap[200] equals 10 for example, it means that the 201^{st} vertex in the vertex buffer has now been moved to the 11^{th} position in the vertex buffer.

The following code snippet shows how we might trim a progressive mesh to subtract 200 triangles from its dynamic range. To do so we decrease maximum face count by 100 triangles and increase the minimum face count by 100 triangles.

```
pMesh->TrimByFaces(pMesh->GetMinFaces()+100,pMesh->GetMaxFaces()-100, NULL, NULL);
```

8.9.7 Progressive Mesh to Standard Mesh Cloning

The ID3DXPMesh interface is derived from ID3DXBaseMesh and as such, inherits ID3DXBaseMesh::CloneMeshFVF (and the declarator version not discussed in this course). This function will allow us to create an ID3DXMesh clone of the progressive mesh at its *current* level of detail. This function is identical to the ID3DXMesh::CloneMeshFVF function covered earlier in the chapter.

This presents an interesting use of the progressive mesh as a one-off simplification utility class for meshes that will not use dynamic LOD in the run-time engine and therefore do not require the rather large simplification step chain to be computed and stored in memory. We can load a standard ID3DXMesh, use it to generate a progressive mesh using the D3DXGeneratePMesh function, set the level of detail to a desired face or vertex count, and then clone the progressive mesh using CloneMeshFVF back out into a vanilla ID3DXMesh. The clone would be a standard ID3DXMesh with a lower level of detail than the original dataset and the progressive mesh interface could be released as we no longer need its simplification abilities.

While there is nothing wrong with using ID3DXPMesh to perform one-off standard mesh simplifications in this way, this is exactly what the ID3DXSPMesh interface is designed to do. It simplifies the mesh in the same way but does not incur any of the memory overhead of the progressive mesh caused by the progressive mesh's runtime simplification system. We will discuss simplification meshes shortly.

The code snippet below demonstrates progressive mesh cloning for mesh simplification. The code assumes that the input mesh has already been validated and has more than 1000 faces to start.

```
ID3DXMesh  *pStandardMesh; // Assume this mesh already contains data
ID3DXPMesh *pPMesh;

D3DXGeneratePMesh(pStandardMesh , pAdjacency , NULL , NULL , 1 ,
                  D3DMESHSIMP_FACE , &pPMesh);

pStandardMesh->Release();
pPMesh->SetNumFaces( 1000 );

pPMesh->CloneMeshFVF( D3DXMESH_MANAGED , pPMesh->GetFVF  ,
                      pDevice , &pStandardMesh);

pPMesh->Release();
```

The result of the above example is a regular ID3DXMesh which has had its face count reduced using a temporary ID3DXPMesh to carry out the simplification task.

8.9.8 Progressive Mesh to Progressive Mesh Cloning

ID3DXPMesh also includes a cloning function (two functions if you count the declarator version) that allows us to clone a progressive mesh to a new progressive mesh. This is useful if we need to make a copy of the progressive mesh and have it remain progressive, or if we would like to change the vertex format of an already progressive mesh, perhaps to make room for a new component.

```
HRESULT ClonePMeshFVF
(
    DWORD Options,
    DWORD FVF,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXPMESH *ppCloneMesh
);
```

The parameter list is identical to the standard cloning function except for the fact that the last parameter must be the address of a pointer to an ID3DXPMesh rather than an ID3DXMesh interface. If the function is successful, the new progressive mesh will contain all of the mesh geometry and collapse structures as the original and it will share the same current LOD state. That is, if we have a 1000 face progressive mesh, reduce its current face count to 100 and then clone it, the clone will also be set to render using 100 faces. Of course, the identical collapse structures and geometry are stored within the clone, so it can always adjust its LOD independently after it is created and can also be adjusted back up to a 1000 face mesh (the maximum detail level of the progressive mesh from which it was cloned).

8.9.9 Progressive Mesh Optimization

The progressive mesh interface exposes two optimization functions, much like the ID3DXMesh. The first, ID3DXPMesh::Optimize creates an optimized standard (non-progressive) mesh whose dataset contains the faces and vertices of the progressive mesh's *current* level of detail. This is exactly like cloning the progressive mesh using the ID3DXPMesh::CloneMeshFVF function to create a standard output mesh and then performing an OptimizeInPlace on it. The original progressive mesh is not altered by this method. This can be a useful way for generating optimized low-poly representations of complex meshes. The function is identical to the ID3DXMesh::Optimize function and as such the parameter list will not be explained again. The function is shown below for your reference.

```
HRESULT ID3DXPMesh::Optimize
(
    DWORD Flags,
    DWORD *pAdjacencyOut,
    DWORD *pFaceRemap,
    LPD3DXBUFFER *ppVertexRemap,
    LPD3DXMESH *ppOptMesh
);
```

ID3DXPMesh does not expose the OptimizeInPlace function for optimization of the current mesh data. This is due in part to the fact that the optimizations that can be performed on the progressive mesh base geometry data are limited by the fact that triangles and vertices have to be in a precise order so that edges can be correctly and speedily collapsed and split. It is still possible for triangles to be reordered in the progressive mesh index buffer in a limited way to optimize for attribute sorting or higher vertex cache hit rates, but the vertex data itself will not be touched. To make this distinction clear, the ID3DXPMesh interface exposes a function to perform an in-place optimization of the progressive mesh base geometry. The name of this function is ID3DXPMesh::OptimizeBaseLOD and it accepts a much smaller parameter list than the ID3DXMesh::OptimizeInPlace function.

```
HRESULT OptimizeBaseLOD
(
    DWORD Flags,
    DWORD *pFaceMap
);
```

DWORD Flags
This is one of the D3DXMESHOPT flags discussed earlier in the chapter (see ID3DXMesh::Optimize or the ID3DXMesh::OptimizeInPlace function). The efficiency of the optimization is limited by the requirements of the progressive mesh system where collapse order takes precedence.

DWORD *pFaceMap
This is a pointer to an array large enough to hold one DWORD for each face in the pre-optimized mesh. When the function returns, it will be filled with information describing how triangles may have moved. This allows us to correct external references if necessary.

8.9.10 Vertex History

Edge collapsing and splitting can result in a visible popping effect as the mesh topology changes suddenly between detail levels. To solve this problem, many 3D engines use a morphing technique to gradually perform the collapse in a controlled fashion over time. As long as the vertex to be collapsed and the neighbor it will collapse onto are known, this is a relatively easy thing to do.

While ID3DXPMesh does not support morphing directly, it does expose a function that can be used to assist us if we want to do it on the application side. ID3DXPMesh::GenerateVertexHistory will fill an application supplied DWORD array (one DWORD per max vertices in the mesh) describing the new index for a vertex collapse.

```
HRESULT GenerateVertexHistory(DWORD *pVertexHistory);
```

For example, if pVertexHistory[500] equals 2 and pVertexHistory[505] equals 2, this tells is that at the current level of detail, vertices 500 and 505 have both been collapsed onto vertex 2.

Morphing techniques are beyond the scope of this chapter, but there are plenty of references in books and on the web if you are interested in researching it. We will examine morphing concepts later in this course series.

8.10 ID3DXSPMesh

The last mesh interface that we will cover in this chapter is the simplification mesh: ID3DXSPMesh. Fortunately, we will be able to cover this interface very quickly since nearly all of the progressive mesh concepts apply to simplification meshes. In fact, we can think of a simplification mesh as a progressive mesh with no dynamic LOD or rendering functionality. In this sense, it is a utility object that is used to apply polygon reduction methods to a standard input mesh to produce a lower detail result. For performance reasons we would not want to use this class for real-time work, but it is quite useful for performing a one-time simplification offline in a tool or at application startup. Perhaps you would use this interface to tailor mesh LOD according to the capabilities of the end user's machine.

Unlike the ID3DXMesh interface and the ID3DXPMesh interface, the ID3DXSPMesh interface is not derived from the ID3DXBaseMesh interface; it is derived directly from IUnknown. This means that the simplification mesh object does not inherit DrawSubset and as such, is not explicitly renderable.

ID3DXSPMesh exposes the ReduceFaces and ReduceVertices functions to simplify the mesh which, once done, are irreversible. Once the mesh is reduced, it must be cloned for rendering. ID3DXSPMesh has two cloning functions (four if you include declarator versions): CloneMeshFVF which clones the simplified data out to a standard output mesh and ClonePMeshFVF which clones the simplified data out to a progressive mesh. In the latter case, the simplified data will serve as the base geometry (maximum LOD) for the progressive mesh. Once cloned, the simplification mesh can be released. It is faster and more memory efficient to use a simplification mesh for one-time reduction because the overhead of building the runtime edge collapse structures for a progressive mesh is avoided.

8.10.1 Simplification Mesh Creation

We create a simplification mesh object using the global D3DX function D3DXCreateSPMesh. Like the progressive mesh, a simplification mesh needs to be created from a standard ID3DXMesh and cannot be loaded from a file or created manually. The parameter list is almost identical to the GeneratePMesh function used to generate progressive meshes.

```
HRESULT D3DXCreateSPMesh
(
    LPD3DXMESH pMesh,
    CONST DWORD *pAdjacency,
    CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights,
    CONST FLOAT *pVertexWeights,
    LPD3DXSPMESH *ppSMesh
);
```

The function accepts a standard ID3DXMesh containing the dataset to simplify. The second parameter points to the input mesh adjacency information. The third and fourth parameters allow us to pass in priorities for vertex component types and for the individual vertices themselves.

The final parameter is the address of a pointer to an ID3DXSPMesh interface to be created on success. As with the generation of a progressive mesh, illegal geometry will cause the function to fail, so we should call D3DXValidMesh and D3DXCleanMesh to repair the mesh in such cases.

The code below shows how we might create a simplification mesh.

```
HRESULT      hRet;
LPD3DXMESH   pStandardMesh;
LPD3DXSPMESH pSimpMesh;
ID3DXBuffer *pAdj;
ID3DXBuffer *pEff;
ID3DXBuffer *pMat;
DWORD        NumMaterials;
DWORD       *pAdjacency;

// Load our standard mesh
hRet = D3DXLoadMeshFromX("Factory.x" ,D3DXMESH_MANAGED, pDevice, &pAdj, &pMat,
                         &pEff, &NumMaterials , &pStandardMesh);

if (FAILED(hRet)) return false;

// We wont use the effects buffer so release it
pEff->Release();

pAdjacency = (DWORD*) pAdj->GetBufferPointer();

// Check to see if mesh is valid
hRet = D3DXValidMesh( pStandardMesh, pAdjacency, NULL );

if (FAILED(hRet))
{
    LPD3DXMESH pCleanedMesh;

    // Attempt to repair the mesh
    hRet = D3DXCleanMesh( pStandardMesh, pAdjacency, &pCleanedMesh, pAdjacency );

    if (FAILED(hRet))
    {
        pMat->Release();
        pStandardMesh->Release();
        pAdj->Release();
        return false;
    }

    // We repaired ok, let's store pointer
    pStandardMesh->Release();
    pStandardMesh = pCleanedMesh;

} // End if invalid mesh

// Generate our simplification mesh object
hRet = D3DXCreateSPMesh( pStandardMesh, pAdjacency, NULL, NULL, &pSimpMesh );
if (FAILED(hRet))
{
    pMat->Release();
```

```
    pStandardMesh->Release();
    pAdj->Release();
    return false;
}

// We're done, release our original mesh
pStandardMesh->Release();
```

8.10.2 Simplification Mesh Usage

We can simplify the mesh data by calling ReduceFaces or ReduceVertices. Both calls accept a single DWORD that describes the target face or vertex count, respectively. This count must be less than or equal to the face or vertex count in the original input mesh.

```
HRESULT ID3DXSPMesh::ReduceFaces( DWORD Faces );
HRESULT ID3DXSPMesh::ReduceVertices( DWORD Vertices );
```

In the case of both of these functions, the exact target face count or vertex count requested may not be able to be met due to certain restrictions.

Continuing the previous code listing, we could request a simplification of the mesh data down to 10 faces using the following code:

```
pSimpMesh->ReduceFaces ( 10 );
```

ID3DXSPMesh also exposes the GetNumFaces and GetNumVertices functions so that we can inquire about the exact face or vertex count after simplification.

```
DWORD ActualNumberOfFaces = pSimpMesh->GetNumFaces();
```

Since ID3DXSPMesh supports no rendering functions, we will need to clone the simplified data out to another mesh type. ID3DXSPMesh::CloneMeshFVF clones the simplified dataset into a standard ID3DXMesh which can then be used for render operations and is identical to this same call in other mesh interfaces.

```
HRESULT ID3DXSPMesh::CloneMeshFVF
(
    DWORD Options,
    DWORD FVF,
    LPDIRECT3DDEVICE9 pDevice,
    DWORD *pAdjacencyOut,
    DWORD *pVertexRemapOut,
    LPD3DXMESH *ppCloneMesh
);
```

ID3DXSPMesh::ClonePMesh is identical to the previous function with the only difference being the final parameter changes to LPD3DXPMESH (a progressive mesh). The maximum level of detail (the base LOD) of the newly created progressive mesh will be the same as the current simplified mesh.

8.10.3 ID3DXSPMesh Misc. Methods

Before we finish up our discussion of the simplification mesh, let us briefly look at the remaining functions exposed by the interface.

HRESULT GetDevice(LPDIRECT3DDEVICE9 *ppDevice)
This function returns a pointer to the device to which the simplification mesh is bound. Notice that we never passed in a pointer to an IDirect3DDevice9 interface when we created the simplification mesh. That is because the owner device is inherited from the standard input mesh.

DWORD GetFVF(VOID)
Allows us to retrieve the FVF flags of the simplification mesh.

DWORD GetMaxFaces(VOID)
Returns the number of faces that exist in the input mesh.

DWORD GetMaxVertices(VOID)
Returns the number of vertices that exist in the input mesh.

DWORD GetOptions(VOID)
Returns a DWORD describing the creation flags for the simplification mesh. What we are really getting here are the creation options for the input mesh since we never explicitly specify creation options when generating a simplification mesh.

HRESULT GetVertexAttributeWeights(LPD3DXATTRIBUTEWEIGHTS *pAttributeWeights*)
Returns the D3DXATTRIBUTEWEIGHTS structure used to perform simplification. This structure describes how much weight each vertex component has when calculating the error metric between two vertices. It will contain either the same values that you passed into the D3DXCreateSPMesh function or a structure containing the default values if you passed NULL. The default values use only the position, weight, and normal of the vertex when calculating vertex collapse errors.

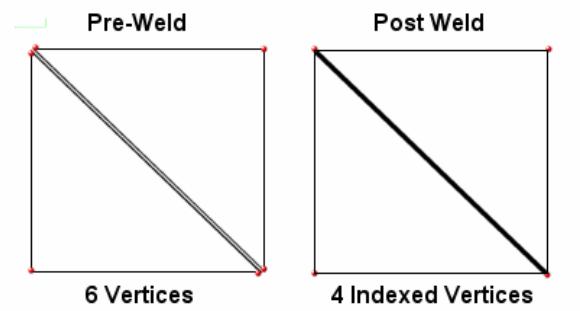
HRESULT GetVertexWeights(FLOAT *pVertexWeights)
Allows you to retrieve the weight of each vertex used to bias the error metric for collapses. If you passed in a per-vertex weights array when you called D3DXCreateSPMesh, then this buffer will be returned with the weight information of each vertex that you passed in. If you passed NULL, then the buffer will be filled with the default weights for each vertex (which is 1.0). This parameter should point to a pre-allocated array that is large enough to hold one DWORD for every vertex in the original/input mesh.

8.11 Global Utility Functions

Before wrapping up this chapter, let us briefly examine some global D3DX functions that can be used with the mesh types we have discussed.

8.11.1 D3DXWeldVertices

This function allows us to weld vertices together in the mesh's vertex buffer. It is essentially another function that can clean and optimize the data for a mesh. This is very useful if the level editor you are using does not generate indices and provides each triangle with its own unique (three) vertices. Consider a quad for example that is stored using 6 vertices (3 for each triangle).



In the case of a D3DX mesh, where all data is stored as indexed triangle lists, the two duplicated vertices at the bottom right and top left corners of the quad could be represented by one vertex (the other discarded) and the indices remapped. Of course, if the two triangles are not supposed to be two halves of the same quad surface and instead had unique textures or colors applied, we definitely would not want to force a vertex weld, as fusing the two vertices into one would cause mesh color/texture distortion.

If two vertices are very close together and share similar or identical properties however, we can use the D3DXWeldVertices function to collapse them into one vertex and reduce the size of our mesh. What this function essentially does is look for vertices that are duplicated and collapse them into a single vertex. This involves removing one of the redundant vertices and changing all indices that reference that vertex to point at the one that remained instead. The function is shown below along with a description of its parameter list.

```
HRESULT D3DXWeldVertices
(
    const LPD3DXMESH pMesh,
    DWORD Flags,
    CONST D3DXWELDEPSILONS* pEpsilon,
    CONST DWORD* pAdjacencyIn,
    DWORD* pAdjacencyOut,
    DWORD* pFaceRemap,
    LPD3DXBUFFER* ppVertexRemap
);
```

const LPD3DXMESH pMesh
This is a standard ID3DXMesh interface pointer to a mesh you want to have welded.

DWORD Flags
A combination of zero or more D3DXMESH flags. Recall that we used these to create a mesh and specify which resource pool the mesh is created in. Since this function does not explicitly generate an output mesh, we can assume that the vertex and index buffers are destroyed and rebuilt because we are able to change resource pools.

CONST D3DXWELDEPSILONS *pEpsilon
To determine whether two vertices should be welded (if they are considered so alike that collapsing them into a single vertex would not significantly alter the appearance of the mesh), a comparison is made between vertex components. If the components match, then the vertices should be welded. To provide tolerance for floating point inaccuracy (ex. 0.998 vs. 0.997), we pass the address of a D3DXWELDEPSILONS structure. It contains a floating point epsilon member for every possible vertex component. This allows us to fine tune exactly how fuzzy the comparisons are when searching for like vertices.

```
typedef struct _D3DXWELDEPSILONS
{
        FLOAT Position;
        FLOAT BlendWeights;
        FLOAT Normal;
        FLOAT PSize;
        FLOAT Specular;
        FLOAT Diffuse;
        FLOAT Texcoord[8];
        FLOAT Tangent;
        FLOAT Binormal;
        FLOAT TessFactor;
} D3DXWELDEPSILONS;
```

Do not concern yourself with the unfamiliar vertex components for now. The vertex structures we will use for the time being require only the components we have seen thus far in the course series.

If we set the Position member to 0.005 and the distance between two vertices is less than 0.005, they will be considered a match (for position). Since we can set tolerance values for each vertex component individually, we might specify a bigger tolerance for the vertex normal test than for the diffuse color. Alternatively, setting the TexCoord[0] member to 0.0 for example means that vertices will only be welded if their first set of texture coordinates are an exact match.

The following code shows how we might fill this structure to perform a weld when each existing vertex component is compared using a tolerance of 0.001.

```
D3DXWELDEPSILONS Epsilons;

 // Set all epsilons to 0.001;
 float *pFloats = (float*)&Epsilons;

 for ( ULONG i = 0; i < sizeof(D3DXWELDEPSILONS) / sizeof(float); i++ )
      *pFloats++ = 1e-3f;
```

CONST DWORD *pAdjacencyIn
This is the adjacency information for the pre-welded mesh.

DWORD* pAdjacencyOut
If we require adjacency information after the mesh has been welded, we can pass a pointer to a DWORD buffer here. It should be large enough to hold 3 DWORDs for every triangle in the pre-optimized mesh.

DWORD* pFaceRemap
LPD3DXBUFFER* ppVertexRemap
These parameters can store face and vertex remapping information to reflect the changes that took place during the operation.

Weld Example:

```
D3DXWELDEPSILONS Epsilons;

// Set all epsilons to 0.001;
float * pFloats = (float*)&Epsilons;
for (ULONG i = 0; i < sizeof(D3DXWELDEPSILONS)/sizeof(float); i++)
     *pFloats++ = 1e-3f;

// Optimize the data
m_Mesh.OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );

// Weld the Data
m_Mesh.WeldVertices( 0, &Epsilons );
```

8.11.2 D3DXComputeNormals

D3DXComputeNormals calculates vertex normals for D3DX mesh objects. Since the function accepts an ID3DXBaseMesh pointer, we can use it to compute the normals for any derived mesh (ID3DXMesh or ID3DXPMesh).

```
HRESULT D3DXComputeNormals
{
    LPD3DXBASEMESH pMesh,
    const DWORD *pAdjacency
);
```

The mesh must have a vertex buffer created with the D3DFVF_NORMAL flag. The per-vertex normal is generated by averaging face normals for faces that share the vertex. If adjacency is provided, replicated vertices are ignored and "smoothed" over. If adjacency is not provided, replicated vertices will have normals averaged only from the faces explicitly referencing them.

This function is extremely useful, especially if you have loaded an X file without vertex normals and would like to add them to the mesh. Simply clone the mesh with the D3DFVF_NORMAL flag set to create room for vertex normals, and then call D3DXComputeNormals to calculate their values.

8.11.3 D3DXSplitMesh

This function is used to split a mesh into multiple meshes. We pass in a source mesh (ID3DXMesh) and the maximum number of vertices allowed per mesh. If the input mesh is larger than the specified maximum size, it will be split into multiple (current / max) ID3DXMesh objects. The new meshes are returned in an ID3DXBuffer as an array of ID3DXMesh pointers.

```
void D3DXSplitMesh
(
    const LPD3DXMESH pMeshIn,
    const DWORD *pAdjacencyIn,
    const DWORD MaxSize,
    const DWORD Options,
    DWORD *pMeshesOut,
    LPD3DXBUFFER *ppMeshArrayOut,
    LPD3DXBUFFER *ppAdjacencyArrayOut,
    LPD3DXBUFFER *ppFaceRemapArrayOut,
    LPD3DXBUFFER *ppVertRemapArrayOut
);
```

const LPD3DXMESH pMeshIn
This is a pointer to the ID3DXMesh interface that will be split into multiple meshes.

const DWORD *pAdjacencyIn
This is a pointer to a buffer containing the input mesh face adjacency information

const DWORD MaxSize
This is the maximum vertex buffer size per mesh.

const DWORD Options
Here we specify zero or more D3DXMESH flags describing concepts like the resource pool for the smaller meshes that are created by this function.

DWORD *pMeshesOut
This DWORD will contain the number of smaller meshes that were created when the function returns. It tells us the number of ID3DXMesh interface pointers in the ppMeshArrayOut buffer.

LPD3DXBUFFER *ppMeshArrayOut
This is the address of an ID3DXBuffer interface pointer that will contain one or more ID3DXMesh interface pointers for the smaller meshes.

LPD3DXBUFFER *ppAdjacencyArrayOut
This buffer will contain face adjacency information for the meshes created. It is an array of N DWORD pointers (one per mesh) that is followed immediately by the actual data (3 DWORDS per face) referenced by these pointers. For example, if we wanted to find the third adjacency DWORD for the fifth mesh we would do the following:

```
DWORD **pAdjacency = (DWORD**)(ppAdjacencyArrayOut->GetBufferPointer());
DWORD   Adjacency = pAdjacency[4][2];
```

LPD3DXBUFFER *ppFaceRemapArrayOut
If we require the face re-map information for each mesh, we can pass the address of an ID3DXBuffer pointer. When the function returns, the buffer will contain the face re-map information (1 DWORD per face) for each mesh created. This is stored in the same way as the adjacency out information described previously. We could retrieve re-map information for the 10^{th} face in the 4^{th} mesh using the following code:

```
DWORD **FaceRemap = (DWORD**)(ppFaceRemapArrayOut->GetBufferPointer());
DWORD   Remap = FaceRemap[3][9];
```

LPD3DXBUFFER *ppVertRemapArrayOut
If we require the vertex re-map information for each mesh, we can pass in the address of an ID3DXBuffer interface pointer. On function return it will contain 1 DWORD per vertex for each mesh created. This information is also stored in the same way as the adjacency and face remap data. We could retrieve the re-map information for the 100^{th} vertex in the 2^{nd} mesh using the following code:

```
DWORD **VRemap = (DWORD**)(ppVRemapArrayOut->GetBufferPointer());
DWORD   Remap  = VRemap[1][99];
```

8.11.4 D3DXSimplifyMesh

As it turns out, there is also a global function to perform one-time mesh simplification. It is an alternative to using the ID3DXSPMesh interface to simplify a standard mesh.

```
HRESULT D3DXSimplifyMesh
(
    LPD3DXMESH pMesh,
    CONST DWORD *pAdjacency,
    CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights,
    CONST FLOAT *pVertexWeights,
    DWORD MinValue,
    DWORD Options,
    LPD3DXMESH *ppMesh
);
```

Most of these parameters should be familiar to you by now. The MinValue parameter is where we pass in the number of vertices or the number of faces we would like the input mesh to be simplified to. This value is interpreted based on the Options parameter, as we saw earlier in the chapter.

The final parameter is the address of a D3DXMesh pointer to store the simplified data. At this point we can release the input mesh and use the output mesh for rendering. This is a quick and easy way to perform one-time simplification at application startup with a single function call.

While this function is very convenient, it does perform the simplification from scratch every time it is called. Therefore, if you wish to clone multiple copies of a simplified mesh, it would be more efficient to use ID3DXSPMesh to perform the simplification once, and then use the interface to clone out the copies.

8.11.5 D3DXIntersect

The D3DXIntersect function determines whether a ray intersects any of the faces in a mesh. This is useful for any number of intersection tasks (collision detection, object picking, etc.). Simply pass in a source mesh and a ray start position and direction vector.

```
HRESULT D3DXIntersect
(
    LPD3DXBASEMESH pMesh,
    CONST D3DXVECTOR3 *pRayPos,
    CONST D3DXVECTOR3 *pRayDir,
    BOOL *pHit,
    DWORD *pFaceIndex,
    FLOAT *pU,
    FLOAT *pV,
    FLOAT *pDist,
    LPD3DXBUFFER *ppAllHits,
    DWORD *pCountOfHits
);
```

LPD3DXBASEMESH pMesh
This is a pointer to the mesh that will be tested for intersection. Since this is a pointer to an ID3DXBaseMesh, we can use this function with both ID3DXMesh and ID3DXPMesh meshes.

CONST D3DXVECTOR *pRayPos
This is a 3D vector describing the model space position of the origin of the ray. For example, if you wanted to test whether a ray from a laser gun has hit a mesh, this vector may initially describe the world position of the gun from which the laser is originating. Since the ray origin must be in the mesh model space, we need to multiply it by the inverse world matrix of the input mesh. If the mesh is already defined in world space, then this step is not necessary since both spaces are the same.

CONST D3DXVECTOR3 *pRayDir
This is a unit length model space direction vector describing the direction the ray is heading with respect to the starting position. If the player look vector was aligned with the world X axis when they fired a laser gun, you might want the laser beam to travel along this same direction. Therefore, you would take the player look vector, multiply it by the inverse of the mesh world matrix to convert it to model space, and then pass it into the function.

BOOL *pHit
If the ray intersects any of the triangles in the mesh, this Boolean will be set to true when the function returns (false otherwise).

DWORD *pFaceIndex
If any faces have been intersected by the ray, this value will contain the zero-based index of the closest triangle that was intersected by the ray. The closest triangle is the triangle that was intersected closest to the ray origin (pRayPos).

FLOAT *pU
If a face has been intersected, this float will contain the barycentric U coordinate for the closest triangle. Barycentric coordinates have two components (U, V) much like texture coordinates. They are also in the range [0, 1]. You can use them to calculate the exact texture coordinate, or diffuse/specular color at the point of intersection because they describe the intersection as how much each vertex is weighted in that intersection. If we have a triangle (v1,v2,v3) then the U coordinate tells us how much v2 is weighted into the result and V tells us how much v3 gets weighted into the result. To find the weight for v1 we simply subtract (v1 = 1.0 – U – V). With these three weights we can interpolate the exact color, texture coordinate, or normal at the exact point of intersection within the face.

FLOAT *pV
If a face has been intersected, this float will contain the barycentric V coordinate for the closest triangle (see previous).

FLOAT *pDist
This is the distance from the ray origin to the intersection point on the closest intersected triangle. Perhaps you would use this distance value to check whether the laser gun in our previous example has hit a face that is considered out of range.

LPD3DXBUFFER *ppAllHits
Because the ray used for intersection testing is infinitely long, it is possible that many faces may be intersected. While the last four parameters return only details for the closest triangle intersected, we can also pass a buffer pointer to be filled with all hits. The function will fill an array of D3DXINTERSECTINFO structures. The D3DXINTERSECTINFO structure contains the face index, barycentric hit coordinates, and a distance from the intersection point to the ray origin. There will be one of these structures in the buffer for each face intersected by the ray.

```
typedef struct _D3DXINTERSECTINFO
{
    DWORD FaceIndex;
    FLOAT U;
    FLOAT V;
    FLOAT Dist;
} D3DXINTERSECTINFO, *LPD3DXINTERSECTINFO;
```

DWORD *pCountOfHits
This parameter will tell us how many faces were hit by the ray (i.e. the number of D3DXINTERSECTINFO structures in the ppAllHits buffer).

8.11.6 D3DXIntersectSubset

We can also run a ray intersection test on specified mesh subsets. This can be used for a wide variety of interesting features. For example, perhaps certain mesh subsets are more vulnerable to attack than others or require a different AI response to collisions. This function is identical to D3DXIntersect except for the fact that it takes the subset attribute ID as an additional parameter.

```
HRESULT D3DXIntersectSubset
(
    LPD3DXBASEMESH pMesh,
    DWORD AttribId,
    const D3DXVECTOR3 *pRayPos,
    const D3DXVECTOR3 *pRayDir,
    BOOL *pHit,
    DWORD *pFaceIndex,
    FLOAT *pU,
    FLOAT *pV,
    FLOAT *pDist,
    LPD3DXBUFFER *ppAllHits,
    DWORD *pCountOfHits
);
```

8.11.7 D3DXIntersectTri

This is a generic ray/triangle intersection function (it does not take a mesh as input). We pass three vectors describing the positions of the triangle vertices and a ray position and direction vector. The triangle is tested against the ray and returns TRUE if the ray intersects, FALSE if not. We can also pass variables for barycentric hit coordinates and distance from the ray origin.

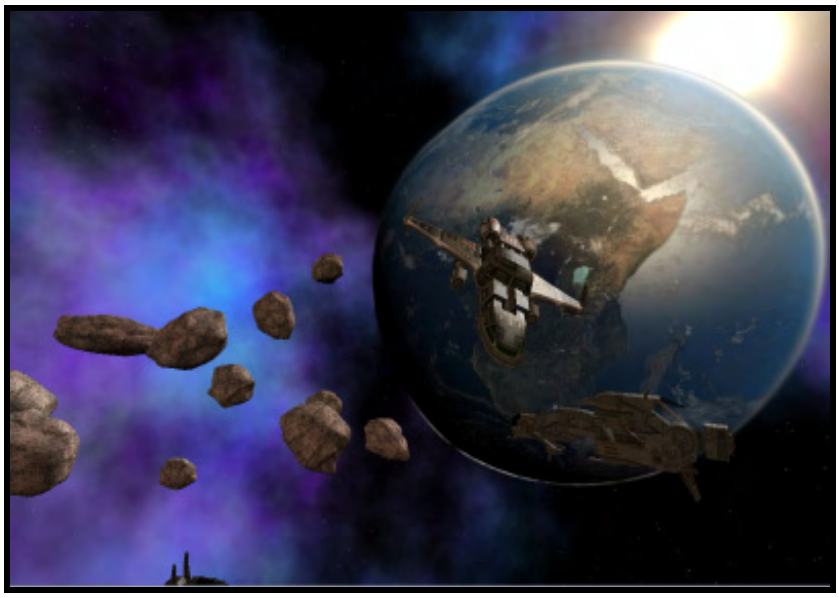
```
BOOL D3DXIntersectTri
(
    const D3DXVECTOR3 *p0,
    const D3DXVECTOR3 *p1,
    const D3DXVECTOR3 *p2,
    const D3DXVECTOR3 *pRayPos,
    const D3DXVECTOR3 *pRayDir,
    FLOAT *pU, FLOAT *pV,
    FLOAT *pDist
);
```

D3DX also provides functions for generating bounding volumes such as spheres and boxes for D3DX mesh objects. These bounding volumes can be used for quick collision detection and frustum culling. We will look at bounding volumes and intersection testing in much more detail a little later in this course.

Conclusion

That wraps up coverage of the three mesh types we wanted to discuss in this chapter. We have certainly seen how D3DX makes what would otherwise be complex coding tasks very simple. In the next chapter we will look at loading mesh hierarchies, which allow us to deal with our scenes in a more organized and flexible fashion. We will also spend a good deal of time exploring the details of the X file format so that we are comfortable with manually parsing such files and creating our own custom templates when needed. This will set the stage for our detailed examination of the D3DX interfaces that support scene animation.

Workbook Eight:
Meshes



Lab Project 8.1: Using ID3DXMesh

The two lab projects in this workbook will be dedicated to the construction of a reusable mesh class. This class will wrap D3DX mesh functionality and employ a sub-system to conveniently handle resource management (the loading and storing of textures and materials). The D3DX library does provide us with an easy means for loading geometry from X files with a single function call, but the burden of loading textures still remains with the application. Our aim is to make this task as easy as possible while avoiding textures being loaded more than once, when they are referenced by more than one mesh.

A mechanism for the efficient rendering of multi-mesh scenes will also need to be put in place. We can not always allow meshes to render themselves independently from the rest of the scene if proper batch rendering is to be achieved. If multiple meshes have subsets that share the same properties, then the scene should be able to intelligently render all subsets from all meshes that share the same device states with a single render call. This will minimize device state changes, an ingredient that is crucial for good performance.

While the D3DX library greatly assists in the loading of X files, if you wish to import your geometry from another source, this has to be accomplished at a lower level. Vertex buffers, index buffers and attribute buffers will need to be locked and manually filled with the data that the application has parsed from a custom file format. An IWF file is an example of such a custom format that we may wish to load and store in a D3DXMesh. This will ensure that the data can benefit from its optimization features. Our wrapper class should expose an interface so that the adding of vertices, indices and attributes is done in as painless a manner as possible. We will expose functions such as AddVertex, AddIndex and AddAttribute which will allow us to build a mesh gradually from data imported from external resources. The vertex buffer, index buffer and attribute buffer will only be constructed internally by the mesh object once all data has been added. This will ensure that we are not locking and unlocking the buffers every time we add a small amount of data (which is highly inefficient).

Finally, as our mesh class is essentially wrapping the underlying ID3DXMesh interface, geometry loaded into our mesh will automatically benefit from the collection of optimization features provided by D3DX.

After the construction of our mesh class you will be able to do the following:

- Populate meshes manually with geometry and attribute data
- Load X files and/or IWF files created by the GILES™ level editor
- Optimize mesh data for rendering
- Create a reusable mesh container class for future applications
- Deal with key aspects of scene state management
    - Texture/material sharing across multiple meshes and their subsets
    - Elimination of redundant texture loading

The CTriMesh Class

The CTriMesh class will be employed as a means for wrapping D3DX mesh functionality. The management of multiple mesh objects and the resources they use will be handled at the scene level. This is made possible by the way each CTriMesh object registers its required resources with the scene texture/material database. To understand this fully, it makes sense for us to look at the CTriMesh class first and later examine the code to the CScene class.

Implementation Goals:

The first design issue to be tackled is texture loading. It does not make sense for the mesh object to manage the loading of textures as the scene or application class is typically the object aware of the limitations of the device. In our demo applications, the scene class is aware of the available texture formats compiled from the device enumeration process. The scene class will therefore be responsible for loading and storing any textures that our mesh objects will use.

Allowing the scene to store and manage the textures used by the mesh is not a trivial design decision. The D3DXLoadMeshFromX function, which our wrapper class will use internally, does not perform any texture loading on our behalf. It returns an array of D3DXMATERIAL structures describing the texture filename and material properties used by each subset in the mesh. We know for example, that the first element in this array contains the texture filename and the material that should be used to render subset 0. The second element in this array contains the attribute properties for subset 1 and so on.

A naive first approach might be to place some code in the mesh wrapper class that, on return from the D3DXLoadMeshFromX function, simply loads all textures referenced in this array. While it is true that every subset in a mesh uses a different texture/material pair, it is probable that multiple subsets in the mesh will use the same texture. In this instance, multiple D3DXMATERIAL structures would contain the same texture filename. Blindly loading the texture referenced by each attribute in this array would cause the same texture to be loaded multiple times. Video memory is a precious resource and we certainly cannot accept this wasteful outcome.

To solve this problem we will expose a function in our mesh class that will allow an external object (our scene class in particular) to register a texture loading callback function. After the scene has instantiated a CTriMesh (but before it has loaded any data into it), it will register a texture loading function with the mesh. The mesh will store a pointer to this function for use when loading data from an X file. When the application calls the CTriMesh::LoadFromX function, the function will first load the geometry using the D3DXLoadMeshFromX function. Then, the mesh object will loop through each of the attributes returned from D3DXLoadMeshFromX and pass each texture filename to the texture loading callback function. The callback (which in our application is a static method of the CScene class), will then check its internal texture list to see if the texture has been previously loaded. If it has, then the function will simply return the texture pointer back to the mesh class for storage. If it has not been loaded, then the CScene class will load the texture using a suitable format and add it to its internal texture database. This pointer is then returned back to the mesh object for storage. After the loading process, a mesh object will thus maintain an attribute array internally which contains one element for each subset in the mesh. Each

element will contain a D3DMATERIAL9 structure and a pointer to an IDirect3DTexture9 interface. The original textures are managed and stored at the scene level.

Using this mechanism, we will solve our redundant texture problem. If multiple subsets use the same texture, their corresponding elements in the internal attribute array will simply contain the same pointer to a texture in the scene texture list. This is true even if multiple meshes use the same texture. They will all contain pointers to the same scene owned texture.

This system provides us with a way to make the mesh essentially self-managed when it comes to rendering since it maintains an internal list of attributes (texture pointer and material) for each subset. This means the mesh object has all the information it needs to render all of its subsets without intervention from the application. The CTriMesh::Draw function will be simple to write. It will loop through each attribute in the attribute array, set the texture and material contained therein, and then use the ID3DXMesh::DrawSubset function to render the appropriate subset in the underlying ID3DXMesh.

This system of self-management makes the mesh class very easy to use. Meshes can be loaded without wasting memory on redundant textures. And, with a single function call, the mesh can be batch rendereded efficiently with mesh triangles that are optimized and sorted by attribute.

This type of self-contained object that requires little external intervention is great for rapid prototyping. It allows an application (such as a mesh viewer) to load and render the mesh with only a few function calls. Each mesh is a self-contained object which will manage its own rendering and therefore is unaware of other mesh objects in the scene. Of course, while easy and intuitive, this approach is not always ideal for all situations, so our mesh class will need to expose an additional mode of operation. The purpose of this additional mode will be to make the mesh more state change friendly when the scene contains multiple meshes.

The importance of batch rendering to minimize state changes has been discussed many times in our studies together. The problem with the self-contained rendering system discussed thus far is that it fails to account for other objects in the scene which may share the same states. It is certainly not uncommon for two or more unique meshes to have cases of identical texture and/or material information at the subset level. For optimal performace, we might want to render all subsets from all meshes which share the same attributes together, to minimize texture and material state changes. This would allow us to batch render across mesh boundaries.

In order for this additional mode (non-managed mode) to allow for state batching across mesh boundaries, the task of rendering the mesh can no longer reside in the mesh class itself. It must now be handled by a higher level object, such as the scene, which can maintain a global list of all attributes (texture and material pairs) used by all meshes in the scene. In this mode, the texture loading callback mechanism will be disabled in favor of another callback function. The new function will also belong to the scene class and will be registered with the mesh. It will accept the complete set of attribute data returned from the D3DXLoadMeshFromX function; not just the texture filename. In this case, the mesh object will not maintain the list of attributes used by each of its subsets; the scene will maintain a global list of attributes used by all meshes.

In order for all meshes to reference the same shared list of attributes, a little remapping of mesh attribute buffers is necessary. This will occur after the mesh is loaded and its textures and materials are registered with the scene. To understand why this is required we must remember that each mesh has its own zero based attribute buffer. While two meshes might each have a subset that both use the same texture and material pair, in the first mesh it may be subset 0 and in the second mesh it may be subset 10.

So we need a way for all of the meshes in the scene to have their attributes remapped. If any meshes have subsets that share the exact same attributes, then we should make sure that both of these subsets use the same attribute ID. An attribute ID will then have global meaning across the entire scene. It is fortunate that attribute IDs within a given mesh do not have to be zero based, or even consecutive, otherwise we would be forced to work with local attributes. If we know that after the remapping process, subset 5 describes the same set of attributes across all meshes, the scene can render the matching subset in all meshes together to reduce state changes. The scene will not loop through each mesh and then render each subset as before, as this would require setting the texture and material states for every subset for every mesh. Instead, the scene will loop through each attribute ID in its internal array and then render any subsets from any mesh that share that attribute together. This way, we set the texture and material for a given scene attribute only once.

The task of remapping the attribute buffer to a global list of attributes is performed when the mesh is first loaded. Our mesh object will pass the attribute data (texture filename and material) returned from the D3DXLoadMeshFromX function to the attribute callback function. This function will search an array of attributes to see if the texture/material data passed in already exists. If so, then its index in the array is returned to the mesh class and becomes the new attribute ID for the subset in question. If a matching attribute cannot be found in the scene attribute array, a new texture/material pair is added to the attribute list and the index of this new element is returned to the mesh. As before, the mesh object can then lock its attribute buffer and alter its contents so that the subset now uses this global attribute ID instead.

To better visualize this process, think of a freshly loaded mesh just about to process the attribute for subset 5. The texture and material used by subset 5 is passed by the mesh into the scene's attribute callback function, where the scene object determines that a matching texture and material already exists in its attribute array at element 35. This means that at least one mesh that was previously loaded used this same attribute combination for one of its subsets. The value of 35 would be returned from this function back to the mesh object. The mesh can now lock the attribute buffer and search for all entries with a value of 5 and replace them with the new global attribute value of 35. Using this method, we can rest assured that all meshes sharing the same attribute data, now share the same global attribute ID for the subsets that reference it.

To summarize, we have achieved some important design goals for our mesh class by envisioning a class that supports two different operational modes:

- ***Managed Mode***: In this mode the mesh object manages its own rendering and setting of states. The mesh maintains an internal attribute array which contains a texture and material pointer for each subset in the mesh. An external texture callback function is used for the loading of texture

filenames returned from the D3DXLoadMeshFromX function. The mesh can be rendered in its entirety with a single function call from the application.

***Pros** : With the exception of the texture callback function, the mesh is self contained. This makes it very easy to setup and render. This mode is ideal for simple applications and prototyping. In this mode the mesh is very easy to plug into existing applications, requiring virtually no support code to be written to access its features. The obvious exception to this is the texture callback function which will need to be implemented and registered with the mesh object if we wish textures to be loaded from the X file.*

***Cons** : The mesh object is not aware of other meshes in the scene and therefore batch rendering across mesh boundaries is not possible. This results in ineffiecient rendering practices when many meshes are in the scene which share matching attributes. This mode is not ideal for using meshes in a typical multi-mesh game environment. Essentially, this mode's greatest strength is in many ways its greatest weakness. By completely encapsulating the rendering of the mesh, the supporting application is denied the chance to use a mesh rendering strategy that it considers optimal.*

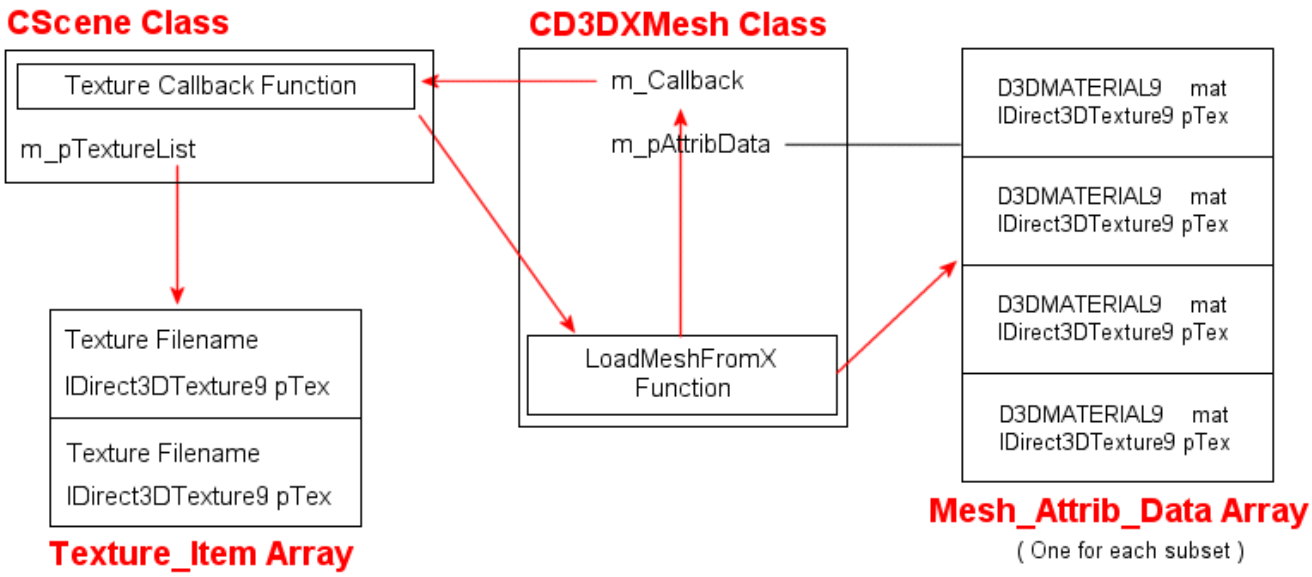
- ***Non-Managed Mode***: In this mode the mesh object no longer maintains an internal list of attributes used by each of its subsets, and is thus incapable of rendering itself. Instead, the meshes pass all attribute data loaded from an X file to an external object (such as our scene class) via an attribute callback function. The scene maintains the attribute list and is responsible for rendering each of the mesh subsets individually. The scene will also be responsible for setting device states before rendering any subset of a given mesh. Attribute data is passed to the scene for registration during X file loading via a callback which returns a global attribute ID for that subset. The mesh will then remap its attribute buffer so that any triangles belonging to the old subset now have attribute IDs matching the newly returned global ID.

***Pros** : Multiple meshes in the scene can have subsets that share the same attribute ID. This attribute ID is actually the index of the texture and material pair stored in the scene's attribute array. The scene can batch render across mesh boundaries. All subsets across all meshes that share the same global attribute ID can be rendered together, minimizing state changes between each render call. This mode is much more efficient for a game scene that contains multiple meshes indexing into a global list of attributes. The mesh object itself is ultimately more flexible in this mode because it does not force any rendering semantics on the application. The application is free to render the various subsets of all meshes in the scene how it sees fit.*

***Cons** : The only real downside for this mode is that more support code must be implemented in the application. This can make integration of the object into new or existing applications a more time consuming affair. The application must supply the rendering logic and maintain and manage the internal attribute list.*

To better visualize the two modes of operation discussed, some diagrams are presented in the next section. They should aid us in seeing the implementations for the mesh in each of these modes.

Managed Mode



Managed mode is the default mode for the class. As seen above, the CScene class maintains an array of Texture_Items. Each texture item contains the filename and texture pointer for a texture that has already been loaded by a mesh in the scene. This array will be empty prior to any meshes having been instantiated.

The CScene class also has a texture callback function which will be registered with the mesh object. This function pointer will be stored in the mesh so that it can be called by the mesh object during X file loading. The CTriMesh::LoadMeshFromX function is exposed to facilitate the loading of X files. This function will initially use the D3DXLoadMeshFromX function to create the underlying D3DXMesh object and initialize its geometry. The mesh object will then loop though each D3DXMATERIAL structure returned from the D3DXLoadMeshFromX function and will pass the texture filename stored there to the texture callback function. If the texture is not already loaded then it will be and added to the end of the scene's Texture_Item array.

The texture callback function will return a texture pointer back to the mesh object to be stored in the mesh's Mesh_Attrib_Data array. Each element in this array contains a material and a texture pointer for a subset in the mesh. The material data is simply copied straight over from the properties returned from D3DX in the D3DXMATERIAL array. The texture pointer is issued to the mesh by the scene object's texture loading callback.

The diagram shows the core components for the mesh in managed mode so we can see that, because the mesh contains its own attribute list, it has everything it needs to render itself. In order for the mesh to render itself in its entirety, simple code can be employed:

```
void CTriMesh::Draw()
{
    // This function is invalid if there is no managed data
    if ( !m_pAttribData ) return;

    // Render the subsets
    for ( ULONG i = 0; i < m_nAttribCount; ++i )
    {
        // Set the attribute data

        pD3DDevice->SetMaterial( &m_pAttribData[i].Material );
        pD3DDevice->SetTexture( 0, m_pAttribData[i].Texture );

        // Otherwise simply render the subset(s)
        m_pMesh->DrawSubset( i );

    } // Next attribute
}
```

The CTriMesh::Draw function will loop through each subset contained in the mesh and manage the setting of the states stored in its attribute array. Then it calls the ID3DXMesh::DrawSubset function, passing in the number of the subset we wish to render. The ID3DXMesh::DrawSubset function renders all triangles in the underlying D3DXMesh with a matching attribute ID.

In this self-contained mode, any burden for rendering the mesh is largely removed from the application. We might imagine that if a scene has many meshes that are all set to operate in managed mode, the code to render those meshes would be a single call to the CTriMesh::Draw method for each mesh in the scene.

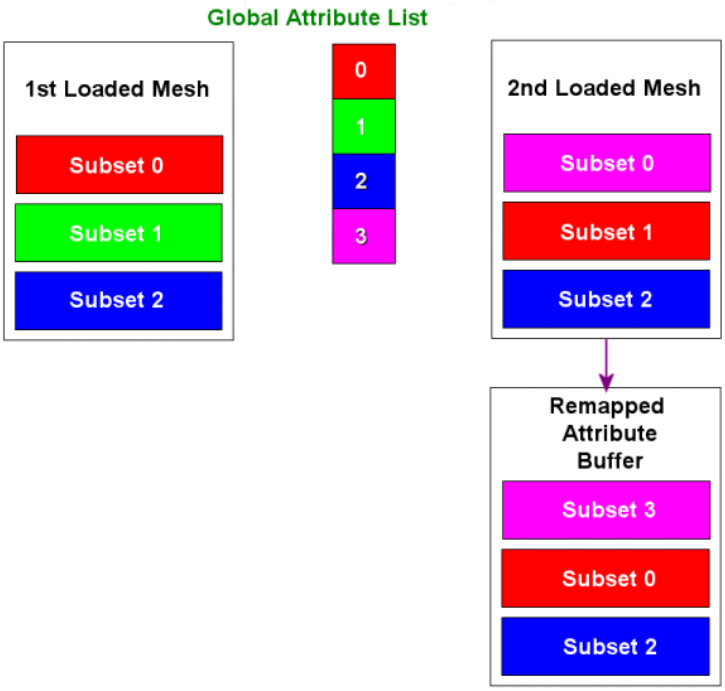
```
for(int i = 0; i < m_NumberOfMeshes; i++)
{
  Meshes[i]->Draw();
}
```

It is the self-contained nature of the managed mode mesh that makes it so quick and easy to plug in and use in existing applications.

Non-Managed Mode

To batch by state across mesh boundaries, we give more rendering control to the scene. In non-managed mode, the mesh does not store an attribute array (this pointer is set to NULL) -- it simply passes the texture/material information loaded from the file to the scene and forgets about it. Subsets will be rendered by the scene using the CTriMesh::DrawSubset function. This function does not update texture or material states and the scene class render method will assume this responsibility. The mesh subset attribute IDs will be remapped to reference a global set of attributes maintained by the scene. This means

that two different meshes that contain matching subset attributes (same texture, same material) can both be remapped to reference the same attribute IDs in the global attribute table. This is demonstrated in the following diagram.

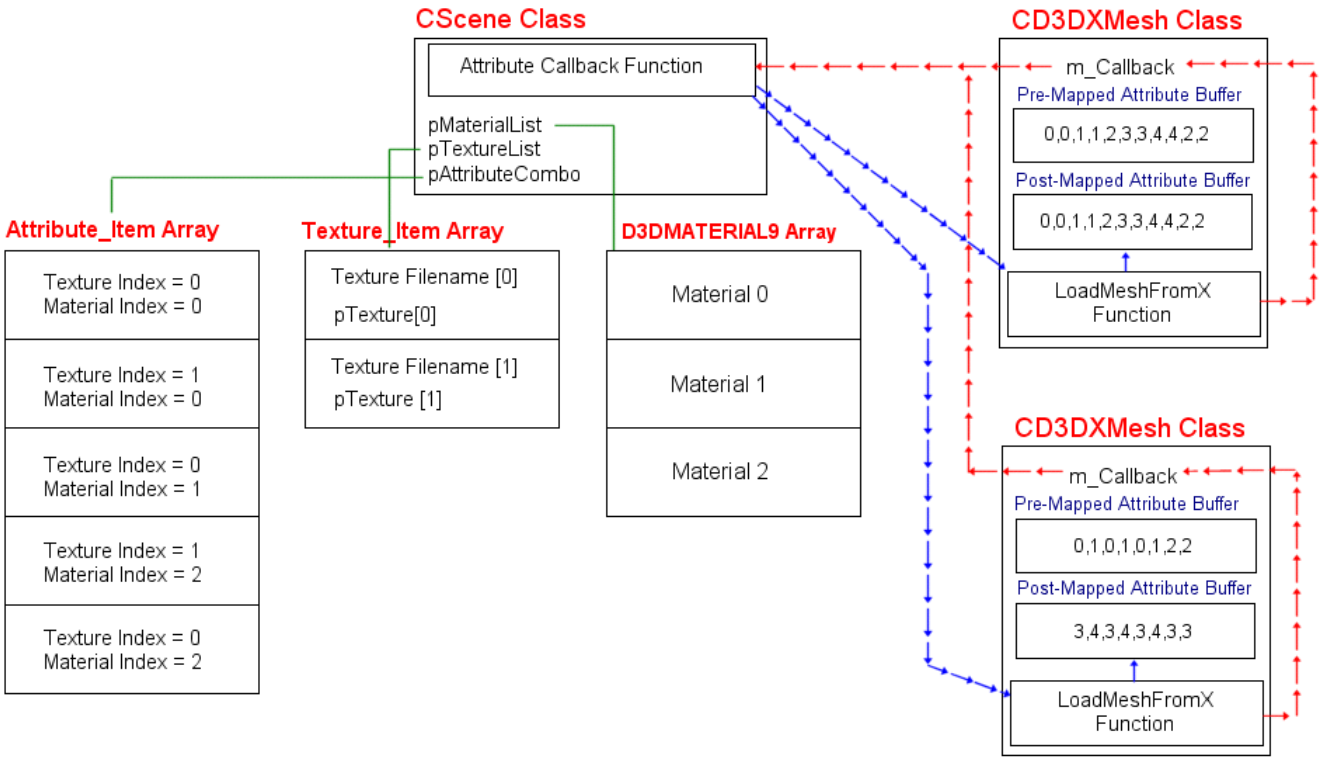


The above image shows how two meshes might initially be loaded. The first and second mesh both have three subsets each. Between both meshes, there are four unique attributes being used. We can see that (using color coding for this example) both the first and second mesh have a red attribute and both also have a blue attribute. The green subset is unique to the first mesh and the purple is unique to the second. We also see that the global attribute list, managed by the scene object, contains those four unique attributes in an ordered array.

While both the first and second mesh contain a red subset, each mesh has a zero based attribute buffer. The attribute buffer values returned from the D3DXLoadMeshFromX function have been assigned differently for the red subset in both meshes. The red material has an attribute ID of 0 in the first mesh and 1 in the second. In order for the scene to render these two subsets together they must both be assigned the same attribute ID. It makes sense to use the index at which the attribute is stored in the global attribute array for this purpose. We can see that the red attribute is stored in this array in position 0. This means any meshes which reference this attribute must use an attribute ID of 0.

The first mesh is basically unaffected because it initially used attribute ID 0 for its red subset. The second mesh however will need to have its attribute buffer locked and altered. Any faces in the red subset of this mesh must have their corresponding attribute ID mapped from 1 (original value when loaded by D3DX) to 0 (the global ID).

The following diagram shows how this model is implemented in our CTriMesh class.



This diagram shows the interaction between the scene and the mesh objects in non-managed mode. The scene class now has to maintain all attribute information used by meshes in the scene; not just textures. The CScene class therefore contains a material array, a texture item array and an attribute combo array. The Texture_Item array is the same one used when dealing with meshes in managed mode. It contans a texture filename, used for testing whether a requested texture has been loaded already, and its accompanying texture pointer. The scene also contains an array of D3DMATERIAL9 structures which will contain all the materials used by all meshes in the scene. These two arrays are maintained by the scene class and logic is put in place to assure that multiple copies of the same texture or material are not loaded more than once. The Attribute_Item_Data array contains what is essentially a scene attribute.

When a non-managed mesh has a subset with an attribute ID of 5 for example, it means the texture and material contained in the 6th element in this array should be used to render that subset. Each element in the scene's attribute array contains a material index and a texture index. These indices are lookups into the Texture_Item array and the D3DMATERIAL9 array. The scene attribute array is analogous to the attribute array that is maintained internally by the mesh object when placed in managed mode. The difference is that this contains the global attribute array for all meshes used in the scene.

In order to place a mesh in non-managed mode we must register an attribute callback function with the mesh, much as we do in managed mode when we register a texture callback function. Registration of this attribute function informs the mesh that it is to be used in non-managed mode. If an attribute callback

has not been registered, the mesh will operate in managed mode and will try to use the texture callback function instead (if one has been registered). The attribute callback function must be registered with the mesh *before* geometry is loaded.

Pay special attention to the two mesh objects shown in the previous diagram as you can see both the pre- and post-mapped attribute buffers. The pre-mapped attribute buffer represents what the mesh looked like just after the D3DXLoadMeshFromX function had been called. Each item is zero based as expected. After the mesh data has been loaded, the mesh object then loops through each attribute and calls the attribute callback function, passing in both the texture and material of each subset. The scene will add the texture and material to its internal arrays and return the index of the corresponding Attribute_Item for that texture/material pair. The attribute buffer of the mesh is then remapped using this information.

Notice how the first mesh that is loaded is also the first to add attributes to the scene. Therefore, the global indices returned for each subset from the callback are the same as the original attribute IDs in the attribute buffer. The second mesh loaded however, has its attribute buffer changed significantly. This is because the attributes it requires were also required by the first mesh and already exist in the scene attribute array.

Once all meshes have been loaded and are known to reference the same global list of attributes, the scene object can render the meshes using whatever method it considers efficient. In the following example we see some code that might be employed by the scene's render function to draw all meshes with minimized texture and material state changes. Batching across mesh boundaries in this fashion is not possible in managed mode.

```
for ( i = 0 ; i < m_TotalNumberOfSceneAttributes; i ++)
{
        pDevice->SetTexture(0, pTextureList[pAttributeCombo[i].TextureIndex]);
        pDevice->SetMaterial(&pMaterialList[pAttributeCombo[i].MaterialIndex]);

        for ( t = 0 ; t < m_NumberOfMeshes ; t ++ )
        {
              m_pMeshes->DrawSubset( i );
        }
}
```

Note that one thing missing from the above code is any mention of the world transform. This is also a device state that must be set.

Often, batching by attribute across mesh boundaries will provide an increase in performance, but this is not always guaranteed. In fact, it is possible that the opposite will happen. If meshes are defined in world space and do not need to be transformed, then batching by attribute across mesh boundaries will provide an increase in performance and is certainly a worthy goal. It is probable however that many of your meshes will be defined in model space and will need to have a world matrix set for them before rendering any of their subsets. If we continue to batch by attribute when this is the case, we are actually increasing the number of times the world matrix has to be set for a mesh. Under normal circumstances, the world matrix would need to be set only once and then the entire mesh rendered. But since attribute

batch rendering requires that we no longer render complete meshes in isolation, the total number of SetTransform calls made per mesh will be equal to its number of subsets:

```
for ( int i=0 ; i<NumberOfAttributes; i ++ )
{
        pDevice->SetTexture ( 0 , pTextureList [ pAttributeCombo[i].TextureIndex]  );
        pDevice->SetMaterial( &pMaterialList  [ pAttributeCombo[i].MaterialIndex] );

        for ( int t =0 ; t < NumberOfMeshes ; t++ )
        {
               pDevice->SetTransform ( D3DTS_WORLD , &pMeshes[t]->Matrix );
               pMeshes[t]->DrawSubset[i];
        }
}
```

SetTransform is a very costly operation and it should not be called superfluously. Usually, the 3D pipeline will try to intelligently predict what information the 3D hardware will need next in its buffers. As a result, the pipeline will often have one or more frames beyond the one you are currently rendering cued up in advance. When the application alters any of its state transform matrices, the pipeline has to flush its buffers. This causes a wait state or stall in the pipeline. In some tests we conducted here in the Game Institute labs, frame rates actually decreased by as much as 50% when batching by attribute on certain scenes because each mesh required multiple world transform sets. This will not always be the case and our non-managed mode mesh provides total flexibility by allowing the scene to choose its preferred rendering strategy. The scene has the ability now to choose how it wishes to batch, by attribute or by transform state.

The scene could choose to render its meshes in the following manner which would assure transform batching over attribute batching. This method would most likely perform better in a scene containing many dynamic meshes.

```
for ( int t = 0 ; t < NumberOfMeshes ; t++ )
{
        pDevice->SetTransform ( D3DTS_WORLD , &pMeshes[t]->Matrix );

        for ( int i=0 ; i<NumberOfAttributes; i ++ )
        {
               pDevice->SetTexture ( 0 , pTextureList [ pAttributeCombo[i].TextureIndex]  );
               pDevice->SetMaterial( &pMaterialList  [ pAttributeCombo[i].MaterialIndex] );

               pMeshes[t]->DrawSubset[i];
        }
}
```

If your scene has many static meshes (which typically lend themselves to being pre-defined in world space), then it would be quicker to batch across mesh boundaries. When a scene consists of both world space meshes and meshes that need a world transform set, you could batch them into two groups and use a different rendering strategey for each group in your core rendering routine.

In our demo applcation we provide the ability for batch render either by attribute or by transform when the mesh objects are in non-managed mode (using a pre-compiler define directive). The scene class

rendering function will test if this define has been set and if so, batch by attribute. Otherwise, batching by transform will be employed.

Source Code Walkthrough

The class declaration for the CTriMesh class is contained in the CObject.h header file. In this listing we have left out the member functions and only show the member variables to improve readability. The class is shown below followed by a description of some of its important member variables.

```
class CTriMesh
{
public:
//-----------------------------------------------------------------------------
// Public Enumerators for This Class.
//-----------------------------------------------------------------------------
 enum CALLBACK_TYPE { CALLBACK_TEXTURE = 0, CALLBACK_EFFECT = 1,
                      CALLBACK_ATTRIBUTEID = 2, CALLBACK_COUNT = 3 };

private:
 //-----------------------------------------------------------------------------
 // Private Variables for This Class
 //-----------------------------------------------------------------------------
    LPD3DXBUFFER  m_pAdjacency;                // Stores adjacency information
    CALLBACK_FUNC m_CallBack[CALLBACK_COUNT];  // References the various callbacks
    LPD3DXMESH    m_pMesh;                     // Physical mesh object
    TCHAR         m_strMeshName[MAX_PATH];       // The filename used to load the mesh


 // Managed Attribute Data
    MESH_ATTRIB_DATA  *m_pAttribData;  // Individual mesh attribute data.
    ULONG              m_nAttribCount; // Number of items in the attribute data array.

 // Mesh creation data.
    ULONG          m_nVertexStride;    // Stride of the vertices
    ULONG          m_nVertexFVF;         // FVF
    ULONG          m_nIndexStride;     // Stride of the indices
    ULONG          m_nVertexCount;     // Number of vertices to use during BuildMesh
    ULONG          m_nFaceCount;       // Number of faces to use during BuildMesh
    ULONG          m_nVertexCapacity;  // We are currently capable of holding this many
                                       // before a grow
    ULONG          m_nFaceCapacity;    // We are currently capable of holding this many
                                       // before a grow
    ULONG          *m_pAttribute;      // Attribute ID's for all faces
    UCHAR          *m_pIndex;          // The face index data
    UCHAR          *m_pVertex;         // The physical vertices.

};
```

LPD3DXBUFFER m_pAdjacency

Unlike ID3DXMesh, the CTriMesh object always maintains a copy of the adjacency information. We use an ID3DXBuffer to store the adjacency data in a uniform and consistent way. Recall that some D3DX functions expect adjacency information as a DWORD array and others expect it as an

ID3DXBuffer. Because the pointer obtained when locking the buffer can be cast to a DWORD pointer, storing it in an ID3DXBuffer makes sure that we account for both cases.

CALLBACK_FUNC m_CallBack[CALLBACK_COUNT]
This is an array which will hold pointers to callback functions. The type CALLBACK_FUNC is a structure defined in the header file Main.h. It contains a void pointer to a function and a void pointer to a context. This second pointer is used to pass arbitrary information to the callback.

```
typedef struct _CALLBACK_FUNC   // Stores details for a callback
{
    LPVOID  pFunction;          // Function Pointer
    LPVOID  pContext;           // Context to pass to the function
} CALLBACK_FUNC;
```

A CTriMesh can hold up to three callback functions for use when loading X files. The first callback (m_CallBack[0]) is used only in managed mode for texture registration. It is where the texture loading callback function pointer will be stored if it has been registered by the application. When the X file is loaded in managed mode, the texture filenames used by each of the mesh attributes will be passed to this function, if it exists. The object that registered the callback with the mesh should make sure that it creates the texture if it does not exist and return a pointer to the texture interface back to the mesh. The returned texture pointer is stored in the mesh MESH_ATTRIB_DATA member. If this callback function is not registered, then no textures will be loaded for the managed mesh. This callback function is not used if the mesh is in non-managed mode and therefore does not need to be registered.

The second array entry (m_CallBack[1]) should contain a pointer to an effect file callback function. X files contain effect file references which the application can use to load the effect file. Effect files are not used in this demo and will be covered in Module III of this series. This callback is only used in managed mode and effect files contained within the X file will be ignored if it is not registered.

The third array entry (m_Callback[2]) essentially switches the mesh into non-managed mode if it contains a function pointer. When this element is not NULL, it should point to an externally defined attribute callback function. For each attribute returned from the D3DXLoadMeshFromX function, the mesh passes in the texture and material to this callback function, if it is defined. The owner of this function should add the material and texture information to its own texture and material lists if they do not already exist and return a new Attribute ID that the mesh class can use to re-map its attribute buffer to point to the global stores. Attribute IDs will then take on global meaning across mesh boundaries and allow for resource sharing. If this function is not registered, then the mesh is assumed to be in managed mode. Registering this function places the mesh in non-managed mode.

There is an enumerated type in the CTriMesh namespace that indexes these callbacks for easy registration:

```
enum CALLBACK_TYPE { CALLBACK_TEXTURE = 0, CALLBACK_EFFECT = 1,
                     CALLBACK_ATTRIBUTEID = 2, CALLBACK_COUNT = 3 };
```

The last member (CALLBACK_COUNT) indicates the total number of callbacks available to the mesh class at present (i.e., it defines the size of the callback array). This allows us to add more callbacks in the future by inserting them into the enum (before the last element) and incrementing CALLBACK_COUNT.

An external object or function that creates a mesh can set the callback function using the CTriMesh::RegisterCallback function:

```
bool   RegisterCallback ( CALLBACK_TYPE Type, LPVOID pFunction, LPVOID pContext );
```

In Lab Project 8.1, the CScene class is responsible for loading the geometry and creating the CTriMesh objects. After a new CTriMesh has been created (but before the data is loaded) the scene registers its callback function as follows:

```
// Create the Mesh
CTriMesh * pNewMesh = new CTriMesh;

// Load in the specified X file
pNewMesh->RegisterCallback(CTriMesh::CALLBACK_ATTRIBUTEID,CollectAttributeID,this);

// Load the X file into the CTriMesh object
pNewMesh->LoadMeshFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

In this example, after we have created the mesh, the scene registers the CALLBACK_ATTRIBUTEID callback function. This places the mesh into non-managed mode. The second parameter is the address of the function that will be called by the mesh for each texture/material combo returned when the X file is loaded. The CScene class includes a function called CScene::CollectAttributeID which accepts a texture and a material and returns a global attribute ID which the mesh can then use to re-map its attribute buffer. This is the also function that is responsible for both creating the texture and adding the texture and the material to the scene database. In this example we do not have to register the texture or effect callback functions since they are only used with managed meshes. Note that the scene passes the ‘this’ pointer so that it is stored in the context and can be passed to the callback function when called from the mesh. We need to do this because the callback function is a static function shared by all instances of the CScene class. As such, the function can only access static variables. The ‘this’ pointer allows us to circumvent that problem as we saw in earlier lessons and access the non-static member variables of the instance of the CScene object being used.

The three callback functions each take different parameter lists and return different values. The function signatures are typedef’d at the top of the file CObject.h and are shown below. If you register any of these callback functions, you must make sure the functions you write have the the exact signatures that the mesh expects.

```
typedef LPDIRECT3DTEXTURE9    (*COLLECTTEXTURE )( LPVOID pContext, LPCSTR FileName );

typedef LPD3DXEFFECT          (*COLLECTEFFECT)(LPVOID pContext,
                                               const D3DXEFFECTINSTANCE & EffectInstance );
```

```
typedef ULONG                (*COLLECTATTRIBUTEID)( LPVOID pContext,
                                                    LPCSTR strTextureFile,
                                                    const D3DMATERIAL9 *pMaterial,
                                                    const D3DXEFFECTINSTANCE * pEffectInstance);
```

Continuing our discussion of the CTriMesh member variable…….

LPD3DXMESH m_pMesh
This is a pointer to the ID3DXMesh wrapped by this class.

TCHAR m_strMeshName[MAX_PATH]
This member will hold the filename of the X file used to load the mesh data.

MESH_ATTRIB_DATA *m_pAttribData
When the mesh is in managed mode (i.e. the attribute callback function has not been registered) the mesh will maintain an array of MESH_ATTRIB_DATA structures -- one for each subset. Each structure describes the texture and material used by the corresponding subset. The structure is defined in the header file CObject.h and is shown below for convenience.

```
typedef struct _MESH_ATTRIB_DATA
{
    D3DMATERIAL9        Material;
    LPDIRECT3DTEXTURE9  Texture;
    LPD3DXEFFECT        Effect;
} MESH_ATTRIB_DATA;
```

When the mesh is in non-managed mode, this pointer will be NULL and no texture and material information will be managed by the mesh.

There are two ways that this array can be populated in managed mode. When using the CTriMesh::LoadMeshFromX function, this array will automatically be populated with the material information in the X file and the texture pointers returned from the texture callback function. If we are building the mesh ourselves, then we use the AddVertex, AddFace and AddAttributeData functions to populate the mesh buffers. The AddAttributeData function makes new space at the end of the MESH_ATTRIB_DATA array (in managed mode) which we can then populate with the material and texture information we desire. When rendering the mesh in managed mode (CTriMesh::Draw), this array is used to set the texture and the material for each subset.

ULONG m_nAttribCount
This value describes how many attributes are in the MESH_ATTRIB_ARRAY. This is only used in managed mode and should always be equal to the number of subsets in the mesh.

ULONG m_nVertexStride
ULONG m_nVertexFVF
ULONG m_nIndexStride
These three members store vertex stride, FVF, and index stride (16 vs. 32-bit) information. Our application will set these as soon as it creates a mesh (before calling any loading functions). We set this information by calling CTriMesh::SetDataFormat:

```
void   SetDataFormat ( ULONG VertexFVF,  ULONG IndexStride);
```

SetDataFormat copies over the information into the internal class variables and uses the specified FVF flags to calculate the vertex stride. This function is only used when creating a mesh from scratch. You must make sure you call it before adding any data to the mesh, otherwise mesh creation will fail. We do not call this function when we create the mesh using CTriMesh::LoadMeshFromX because the vertex and index formats will be taken from the X file. Recall that the D3DXLoadMeshFromX function will set the FVF of our vertices when the mesh is created based on the vertex components available in the X file.

UCHAR *m_pVertex
ULONG m_nVertexCount
We simplify the manual creation of mesh data by maintaining temporary system memory vertex, index, and attribute arrays. This allows an application to accumulate data in these arrays and only build the actual ID3DXMesh once all arrays are finalized. Once the mesh is created, these arrays can be freed since the data is no longer needed. The m_nVertexCount member describes the current number of vertices in the temporary array and does not necessarily describe the number of vertices in the underlying mesh. This is especially true after the underlying mesh has been optimized. These variables are not used at all if we create the mesh data using CTriMesh::LoadMeshFromX.

UCHAR *m_pIndex
ULONG *m_pAttribute
ULONG m_nFaceCount
These are also temporary storage bins that will be used during manual mesh filling. m_pIndex will contain three indices per face (m_nFaceCount*3) and the m_pAttribute array will hold m_nFaceCount DWORD values. These are the per-face attribute IDs. These members are not used if the mesh data is created using the CTriMesh::LoadMeshFromX function. When we manually add faces to the mesh we also specify the attribute ID of the face about to be added. The indices and attributes passed for each face are stored in these temporary bins and later used to populate the actual index and attribute buffers of the ID3DXMesh.

ULONG m_nVertexCapacity
ULONG m_nFaceCapacity
These member variables are only used during manual mesh creation. When we use the AddVertex method to add a vertex to the m_pVertex array, the array will need to be resized to accommodate the new addition. Array resizing can be slow, so to avoid doing it for single vertex additions, we use the m_nVertexCapacity member to define a resize threshold. For example, the array will be resized only

after every 100 vertices have been added. If we start with a vertex capacity of 100, we only need to resize the array when we add the 101st vertex. We then resize again by 100 and increase m_nVertexCapacity by 100 also. We can then once again add another 100 vertices before the new vertex capacity of 200 is reached. The vertex array is resized again by 100, and so on. While it might seem wasteful to resize the array by 100 when we may only need to add 1 vertex, this memory is freed as soon as the underlying ID3DXMesh object is created. Thus the memory footprint is only temporary and it does greatly speed up manual mesh creation. m_nFaceCapacity works in precisely the same way but with the index and attribute arrays. These member variables are not used once the underlying ID3DXMesh has been created.

The Methods

Many of the CTriMesh functions are just wrappers around their ID3DXMesh counterparts (e.g., the functions which optimize and render the underlying mesh data). Some may be slightly more complex than others due to the management modes the class supports. This is certainly true in the managed mode code sections where asset management code is introduced to cater for its self-contained design. We will look at the list of the methods next, but only discuss source code for non-wrapper functions where the code inside the function is new or significant.

CTriMesh::CTriMesh()

The mesh class has a default constructor that takes no parameters and initializes all internal variables to NULL or zero.

```
CTriMesh::CTriMesh()
{
    // Reset Variables
    m_pAdjacency      = NULL;
    m_pMesh           = NULL;
    m_pAttribData     = NULL;
    m_nAttribCount    = 0;
    m_nVertexCount    = 0;
    m_nFaceCount      = 0;
    m_pAttribute      = NULL;
    m_pIndex          = NULL;
    m_pVertex         = NULL;
    m_nVertexStride   = 0;
    m_nIndexStride    = 0;
    m_nVertexFVF      = 0;
    m_nVertexCapacity = 0;
    m_nFaceCapacity   = 0;

    ZeroMemory( m_strMeshName, MAX_PATH * sizeof(TCHAR));

    // Clear structures
    for(ULONG i = 0; i < CALLBACK_COUNT; ++i )
        ZeroMemory( &m_CallBack[i], sizeof(CALLBACK_FUNC) );
}
```

Notice how CALLBACK_COUNT is being used to initially clear the function callback array. Initially no callback functions are registered so the mesh is assumed to be in managed mode at this point. However, textures and effect files would be ignored during loading unless a callback is given.

CTriMesh::~CTriMesh()

The destructor calls CTriMesh::Release to free the internal memory used by the object.

```
CTriMesh::~CTriMesh()
{
    Release();
}
```

CTriMesh::Release

The Release function manages the freeing of memory resources used by the mesh. This allows us to release the underlying data and reuse the same CTriMesh object to create another CTriMesh object if desired. This function calls Release for any COM objects it is currently managing so that reference counts are decremented as expected. This allows the COM layer to unload those objects from memory if no other references to those objects remain outstanding.

If the mesh has an internal attribute array defined (which is only the case if the mesh is operating in managed mode), then the reference count of any textures and effect files contained therein are decremented and the attribute array deleted. It is likely that the texture objects will not be released from memory at this point since the scene object will also have a reference to this object. The same texture resource may also be referenced by other meshes in the scene. Only when all meshes have released their claim to a texture and the scene itself releases it, will the reference count hit zero and the texture be unloaded from memory. It is especially important that our mesh class abides by the COM reference counting convention and is well behaved in this regard. If we did not make sure that the reference count of a texture was increased and decreased correctly when a mesh object gains or releases its claim to a texture resource, we might end up with a scenario where meshes in the scene still have pointers to textures which no longer exist.

```
void CTriMesh::Release( )
{
    // Release objects
    if ( m_pAdjacency ) m_pAdjacency->Release();
    if ( m_pMesh      ) m_pMesh->Release();

    // Release attribute data.
    if ( m_pAttribData )
    {
        for ( ULONG i = 0; i < m_nAttribCount; i++ )
        {
            // First release the texture object (addref was called earlier)
            if ( m_pAttribData[i].Texture ) m_pAttribData[i].Texture->Release();
```

```
            // And also the effect object
            if ( m_pAttribData[i].Effect  ) m_pAttribData[i].Effect->Release();

        } // Next Subset

        delete []m_pAttribData;

    } // End if subset data exists

    // Release flat arrays
    if ( m_pVertex    ) delete []m_pVertex;
    if ( m_pIndex     ) delete []m_pIndex;
    if ( m_pAttribute ) delete []m_pAttribute;

    // Clear out variables
    m_pAdjacency      = NULL;
    m_pMesh           = NULL;
    m_pAttribData     = NULL;
    m_nAttribCount    = 0;
    m_nVertexCount    = 0;
    m_nFaceCount      = 0;
    m_pAttribute      = NULL;
    m_pIndex          = NULL;
    m_pVertex         = NULL;
    m_nVertexStride   = 0;
    m_nIndexStride    = 0;
    m_nVertexFVF      = 0;
    m_nVertexCapacity = 0;
    m_nFaceCapacity   = 0;

    ZeroMemory( m_strMeshName, MAX_PATH * sizeof(TCHAR));
}
```

Note that we do not reset the callback array to zero. It may well be the case that you want to free the internal data but re-use the class to create another mesh. In this case, you will probably want to use the previously registered callback functions to load a new mesh.

Loading Mesh Data

Once we have a new CTriMesh object, there are two ways to populate it. We can either load the data from an X file using CTriMesh::LoadMeshFromX or we can use the AddVertex, AddFace, and AddAttributes functions to add the data to the mesh manually. In the latter case we would call CTriMesh::BuildMesh to create the final ID3DXMesh. Because the two methods of mesh population are quite different, we will cover them in two sections.

A. Loading Data from X Files

If we intend to load the mesh from an X file, then before calling CTriMesh::LoadMeshFromX we will want to register one or more callback functions to handle asset management. The following example demonstrates texture callback registration for a managed mode mesh. Registering this function will have no effect for a non-managed mode mesh since the attribute callback function will be called instead in that case.

```
CTriMesh MyMesh;
MyMesh.RegisterCallback ( CTriMesh::CALLBACK_TEXTURE , MyFunction ,  pSomeInfo );
```

pSomeInfo is a pointer to arbitrary data (or NULL) that we wish the mesh class to pass to the callback function when it is called. We will look at the code to this function next.

CTriMesh::RegisterCallback

The mesh class allows for registration of three types of callback functions to handle asset management requirements. When instructed to load an X file, the mesh can use these functions to pass texture names and material properties to the external object that registered the callback. In our application, the CScene class is responsible for loading and storing textures and making sure that we do not load redundant texture copies.

```
bool CTriMesh::RegisterCallback( CALLBACK_TYPE Type, LPVOID pFunction, LPVOID pContext )
{
    // Validate Parameters
    if ( Type > CALLBACK_COUNT ) return false;

    // You cannot set the functions to anything other than NULL
    // if mesh data already exists (i.e. it's too late to change your mind :)
    if ( pFunction != NULL && m_pAttribData ) return false;

    // Store function pointer and context
    m_CallBack[ Type ].pFunction = pFunction;
    m_CallBack[ Type ].pContext  = pContext;

    // Success!!
    return true;
}
```

This function is passed a member of the CALLBACK_TYPE enumerated type (CALLBACK_TEXTURE, CALLBACK_EFFECT or CALLBACK_ATTRIBUTEID) indicating the callback function being set. The first two types are used if the mesh is in managed mode. Registering the CALLBACK_ATTRIBUTEID function places the mesh object into non-managed mode. When in non-managed mode we do not need to register the texture or effect callback functions as they will never be called. Note that CALLBACK_TYPE also serves as the callback array index of each function pointer.

The second parameter is a pointer to the callback function. It will be stored in the callback array. The third parameter allows the external object that is registering the callback to store arbitrary data that it wants passed to the callback function on execution. Because our callback functions are static, the CScene class will pass the ‘this’ pointer as the context to ensure access to non-static members.

CTriMesh::LoadMeshFromX

Once we have registered our callback functions, we call LoadMeshFromX to populate the mesh with data from an X file. The following example shows how this function might be called. Again, it is important to call this function only after callback registration so that attribute data is handled properly.

```
MyMesh.LoadMeshFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

The function takes three parameters: a string containing the filename of the X file, a DWORD containing our desired mesh creation flags, and a pointer to the device that will own the mesh resources. The mesh creation flags are a combination of zero or more D3DXMESH flags that we use when creating an ID3DXMesh. These flags are passed straight to the D3DXLoadMeshFromX function to control properties such as the memory pools used for the vertex and index buffers.

Note as well that we do not need to specify vertex stride, FVF, or index size since the X file will provide that information. If the end result is not to our liking, we can always clone the CTriMesh to a different format after it has been loaded.

> **Note:** Due to the lengthy nature of this function, we will leave out error checking to compact the listing. The complete version can be found in the source code that accompanies this lesson.

```
HRESULT CTriMesh::LoadMeshFromX( LPCSTR pFileName, DWORD Options, LPDIRECT3DDEVICE9 pD3D )
{
    HRESULT      hRet;
    LPD3DXBUFFER pMatBuffer;
    LPD3DXBUFFER pEffectBuffer;
    ULONG        AttribID, i;
    ULONG        AttribCount;
    ULONG       *pAttribRemap  = NULL;
    bool         ManageAttribs  = false;
    bool         RemapAttribs   = false;
```

The first thing the function does is allocate some local variables. It initially assumes the mesh is in non-managed mode by setting the ManageAttribs Boolean to false. We also declare two ID3DXBuffer interface pointers that will passed into D3DXLoadMeshFromX to be filled with material and effect data from the X file.

Since it is possible that the user may have called this function for an object that already has mesh data defined, we call the Release member function (shown earlier) to free up any potential old data.

```
    // Release any old data
    Release();
```

Next we call D3DXLoadMeshFromX with the appropriate parameters (filename, options, etc.). Since we will maintain adjacency data, we pass in the module level CTriMesh::m_pAdjacency member. If the call is successful, the final mesh will be stored in the class variable CTriMesh::m_pMesh.

```
    // Attempt to load the mesh
    D3DXLoadMeshFromX( pFileName, Options, pD3D, &m_pAdjacency, &pMatBuffer,
                       &pEffectBuffer, &AttribCount, &m_pMesh );
```

At this point the mesh has been created and it contains the geometry from the X file. The remainder of the function processes the material and texture information loaded from the file and returned from the D3DXLoadMeshFromX function. In the above code, we can see that it is the pMatBuffer variable which will contain an array of D3DXMATERIAL structures, one for each subset in the mesh. The AttributeCount local variable will contain the number of subsets in the mesh. How we process and store this returned attribute data from this point on depends on whether the mesh is in managed or non-managed mode.

The first thing we do is test wheter the CALLBACK_ATTRIBUTEID callback function is NULL. If it is, then the non-managed mode callback function has not been registered, and this is a managed mesh. In this instance we set the ManageAttribs Boolean to true. Next we store the number of subsets in this mesh in the m_nAttribCount member variable.

```
    // Are we managing our own attributes ?
    ManageAttribs = (m_CallBack[ CALLBACK_ATTRIBUTEID ].pFunction == NULL);
    m_nAttribCount = AttribCount;
```

If this is a managed mesh, then we need to allocate an array of MESH_ATTRIB_DATA structures -- one for each subset. They will be filled with the texture and material information for each subset so that the mesh can render itself using the CTriMesh::Draw function. Once the array is allocated, we initialize it to zero as shown below.

```
    // Allocate the attribute data if this is a manager mesh
    if ( ManageAttribs == true && AttribCount > 0 )
    {
        m_pAttribData  = new MESH_ATTRIB_DATA[ m_nAttribCount ];
        ZeroMemory( m_pAttribData, m_nAttribCount * sizeof(MESH_ATTRIB_DATA));
    } // End if managing attributes
```

If this is a **non**-managed mesh then we will not need to allocate an attribute array as no textures or materials will be stored in the mesh itself. We will however, potentially need to re-map the attribute IDs in the D3DXMesh attribute buffer so that they index into a global pool of resources at the scene level. Therefore, we will allocate a temporary array that will hold new attribute IDs for each mesh subset.

```
    else
    {
        // Allocate attribute remap array
        pAttribRemap = new ULONG[ AttribCount ];

        // Default remap to their initial values.
        for ( i = 0; i < AttribCount; ++i ) pAttribRemap[ i ] = i;
    }
```

By default we initialize each new attribute ID so that it matches the original attribute ID. If the mesh had five subsets for example, this array will be five DWORDS long and each element will initially contain 0

to 4 respectively. This array will be used to store the new values that each ID will have to be mapped to, which will be determined momentarily. Thus if pAttribRemap[2] = 90 after the attribute callback function has been called, this means that the third subset of this mesh uses the texture and material combination that is stored in the 91st position in the scene's global attribute list. We would then loop through the attribute buffer of the ID3DXMesh and change all attribute ID's that currently equal 2 to the new value 90.

Next we retrieve pointers to the material and effects buffers returned from the D3DXLoadMeshFromX function. Although this mesh supports effect instance parsing, we will not be using it in this lesson and effects can be ignored for the time being. Notice how we lock the buffers and cast the pointers to the correct type to step through the data in each buffer.

```
    // Retrieve data pointers
    D3DXMATERIAL       *pMaterials = (D3DXMATERIAL*)pMatBuffer->GetBufferPointer();
    D3DXEFFECTINSTANCE *pEffects   = (D3DXEFFECTINSTANCE*)pEffectBuffer->GetBufferPointer();
```

Now we can loop through each subset/attribute and parse the material data. This is handled differently depending on whether the mesh is in managed or non-managed mode. We will look at the managed case first.

```
    // Loop through and process the attribute data
    for ( i = 0; i < AttribCount; ++i )
    {
```

The material buffer contains D3DXMATERIAL structures that store a D3DMATERIAL9 structure and a texture filename. First we copy the D3DMATERIAL9 into the correct slot in our MESH_ATTRIB_DATA array. For example, if we are currently processing subset 5 in the loop, then we will copy the material into m_pAttribData[4].Material. Since the material information in an X file does not contain an ambient property, we manually set the ambient member to full reflectance (1.0f, 1.0f, 1.0f, 1.0f) after the copy. You should feel free to change this default behavior.

```
        if ( ManageAttribs == true )
        {
            // Store material
            m_pAttribData[i].Material = pMaterials[i].MatD3D;

            // Note : The X File specification contains no ambient material property.
            //        We should ideally set this to full intensity to allow us to
            //        control ambient brightness via the D3DRS_AMBIENT renderstate.
            m_pAttribData[i].Material.Ambient = D3DXCOLOR( 1.0f, 1.0f, 1.0f, 1.0f );
```

To process the texture filename stored in the D3DXMATERIAL buffer for this subset we will use the registered texture callback function. The following code shows how we define a function pointer called CollectTexture to point at the texture callback function in the mesh's callback function pointer array.The CollectTexture variable is a pointer of type COLLECTTEXTURE which we saw defined earlier as a pointer to a function with the desired function signature.

```
            // Request texture pointer via callback
            if ( m_CallBack[ CALLBACK_TEXTURE ].pFunction )
            {
                COLLECTTEXTURE CollectTexture =
                                (COLLECTTEXTURE) m_CallBack[ CALLBACK_TEXTURE ].pFunction;
```

We now call the function using this pointer, passing in the context data that was set for this callback when it was registered. In our case this will be a pointer to the instance of the CScene class that registered the function. We also pass the texture filename for the current subset being processed. The function should return a pointer to an IDirect3DTexture9 interface that will be copied into the Texture member of the mesh attribute array for this subset. In keeping with proper COM standards, we increase the interface reference count of this texture pointer as we make a copy of it.

```
               m_pAttribData[i].Texture = CollectTexture(
                                                m_CallBack[ CALLBACK_TEXTURE ].pContext,
                                                pMaterials[i].pTextureFilename );

                // Add reference. We are now using this
                if ( m_pAttribData[i].Texture ) m_pAttribData[i].Texture->AddRef();

            } // End if callback available
```

Next we need to process the CALLBACK_EFFECT callback function using the same process. We will not register an effect callback function in our application, but we have included the code for future use. When the effect callback function is registered, we pass the context and the D3DXEFFECTINSTANCE structure for this subset. This information was returned from the D3DXLoadMeshFromX function in the pEffects buffer. The effect callback function returns a pointer to an ID3DXEffect interface which is stored in the mesh attribute data array for the current subset. You can feel free to ignore the code that deals with effects until we get to Module III.

```
            // Request effect pointer via callback
            if ( m_CallBack[ CALLBACK_EFFECT ].pFunction )
            {
                COLLECTEFFECT CollectEffect =
                                (COLLECTEFFECT)m_CallBack[ CALLBACK_EFFECT ].pFunction;
                m_pAttribData[i].Effect =
                      CollectEffect( m_CallBack[ CALLBACK_EFFECT ].pContext, pEffects[i] );

                // Add reference. We are now using this
                if ( m_pAttribData[i].Effect ) m_pAttribData[i].Effect->AddRef();

            } // End if callback available

        } // End if attributes are managed
```

At the end of this loop, the managed mesh will have all texture and material information used by each subset stored in the mesh attribute data array. Thus, it can render each of its subsets (including state setting) in a self-contained manner.

If this is not a managed mesh then the chain of events is quite different. We no longer have to allocate a mesh attribute data array and we will not store the texture and material information inside the object.

Management of textures and materials is left to the caller. In order to do this, we must pass the texture filename, the material, and the effect instance to the CALLBACK_ATTRIBUTEID callback function. This function will belong to the CScene class, which maintains a list of all texture and material combinations used by all meshes in the scene. The callback will search its list for a matching attribute (i.e. texture and material combination) and if found, will return the index of this attribute set in its attribute list. If the attribute does not exist in the global list, it will be created and appended. The result is the same from the mesh's perspective -- an index is returned. We store this index in our temporary re-map array so that we can update the subset attribute ID to this new index. After the loop is complete, a non-managed mesh will have an array of re-map values describing what each attribute ID in the mesh attribute buffer should be re-mapped to.

```
        else
        {
            // Request attribute ID via callback
            if ( m_CallBack[ CALLBACK_ATTRIBUTEID ].pFunction )
            {
                COLLECTATTRIBUTEID CollectAttributeID =
                          (COLLECTATTRIBUTEID)m_CallBack[ CALLBACK_ATTRIBUTEID ].pFunction;

                AttribID = CollectAttributeID( m_CallBack[ CALLBACK_ATTRIBUTEID ].pContext,
                                               pMaterials[i].pTextureFilename,
                                               &pMaterials[i].MatD3D,
                                               &pEffects[i] );

                // Store this in our attribute remap table
                pAttribRemap[i] = AttribID;

                // Determine if any changes are required so far
                if ( AttribID != i ) RemapAttribs = true;

            } // End if callback available

        } // End if we don't manage attributes

    } // Next Material
```

We no longer need the material and effect buffers that were returned by D3DXLoadMeshFromX because the callbacks have taken care of loading and storing the resources. Therefore, they can be released.

```
    // Clean up buffers
    if ( pMatBuffer    ) pMatBuffer->Release();
    if ( pEffectBuffer ) pEffectBuffer->Release();
```

If this is a non-managed mesh and attributes require remapping, then we lock the attribute buffer, update it with the new information, and unlock. We can release the remapping data when we are finished because the modifications have now been made to the attribute buffer of the D3DXMesh.. Remember that this buffer contains an attribute ID for each triangle in the mesh.

```
    // Remap attributes if required
    if ( pAttribRemap != NULL && RemapAttribs == true )
    {
        ULONG * pAttributes = NULL;
```

```
        // Lock the attribute buffer
        m_pMesh->LockAttributeBuffer( 0, &pAttributes );

        // Loop through all faces
        for ( i = 0; i < m_pMesh->GetNumFaces(); ++i )
        {
            // Retrieve the current attribute ID for this face
            AttribID = pAttributes[i];

            // Replace it with the remap value
            pAttributes[i] = pAttribRemap[AttribID];

        } // Next Face

        // Finish up
        m_pMesh->UnlockAttributeBuffer( );

    } // End if remap attributes

    // Release remap data
    if ( pAttribRemap ) delete []pAttribRemap;
```

Finally, we store the filename of the X file that was loaded in the m_strMeshName string and return success.

```
    // Copy the filename over
    _tcscpy( m_strMeshName, pFileName );

    // Success!!
    return S_OK;
}
```

The overview of the loading process is not complete until we look at the code for the CScene class callback functions. However, the above function is all that is needed to handle resource management from the perspective of the mesh object. We will look at the CScene callback functions after we have finished examining the methods of CTriMesh.

B. Loading Mesh Data Manually

We have covered the basic loading operations involved for managed and non-managed meshes extracted from X files. Let us now examine the member functions used to manually populate a CTriMesh with vertex, index, and attribute data. An application would need to do this when the mesh data is being created programmatically. More importantly, manual mesh creation will be necessary when the application is loading data from an alternative file format (e.g., IWF, 3DS, etc.). In this section, we will discuss the manual creation methods in the order in which they are normally used so that we get a better feel for the flow of operations.

CTriMesh::SetDataFormat

When we load a mesh from an X file, D3DXLoadMeshFromX chooses the format of our vertices and indices based on the matching data stored in the X file. When creating a mesh manually, before we add any vertex or index data, we must inform the CTriMesh object of the vertex and index formats we will be using. This is the only way it will know how to create the ID3DXMesh. To do this, we call the CTriMesh::SetDataFormat function. This is usually the first function we will call after we have instantiated a mesh object.

```
void CTriMesh::SetDataFormat( ULONG VertexFVF, ULONG IndexStride )
{
    // Store the values
    m_nVertexFVF    = VertexFVF;
    m_nVertexStride = D3DXGetFVFVertexSize( VertexFVF );
    m_nIndexStride  = IndexStride;
}
```

The first parameter is the FVF flag, describing the vertex format for the mesh. The second parameter is the size of the indices (in bytes) we wish to use. This will either be 2 or 4 depending on whether we wish to use 16-bit or 32-bit indices. This function calls the global D3DX helper function D3DXGetFVFVertexSize which returns the size (stride) of a single vertex given the FVF flag passed in.

The following code snippet demonstrates how we might instantiate a mesh which we intend to use for manual data population and set its vertex and index formats:

```
CTriMesh MyMesh;
DWORD  FVFFlags = D3DFVF_XYZ  | D3DFVF_TEX1;
MyMesh.SetDataFormat ( FVFFlags , 2 );
```

Here we are creating a mesh that will have room for a 3D position vector and a single set of 2D texture coordinates stored in each vertex. We also inform the mesh object that we intend to use 16-bit indices.

CTriMesh::AddVertex

This function is used to add vertices to a temporary vertex array maintained by the mesh. It is very much like the functions we have studied in previous lessons that manage array allocation and resizing. In our application, the vertex capacity for the array is initially set to 100 and is increased by 100 every time the limit is reached. This speeds up adding the vertex data to the array and minimizes memory fragmentation. This function adds no data to the ID3DXMesh, as it has not even been created yet. All we are doing is adding vertices to a temporary vertex array which will later be used to create and populate the vertex buffer of the ID3DXMesh. Once the actual ID3DXMesh has been created, this array will no longer be used and may be released.

The first parameter describes the number of vertices we wish to add and the second parameter is a pointer to one or more vertices. We use a void pointer for the vertex data because it must be able to handle arbitrary vertex formats with varying sizes.

```
long CTriMesh::AddVertex( ULONG Count , LPVOID pVertices   )
{
    UCHAR * pVertexBuffer = NULL;

    if ( m_nVertexCount + Count > m_nVertexCapacity )
    {
        // Adjust our vertex capacity (resize 100 at a time)
        for ( ; m_nVertexCapacity < (m_nVertexCount + Count) ; ) m_nVertexCapacity += 100;

        // Allocate new resized array
        if (!(pVertexBuffer = new UCHAR[(m_nVertexCapacity) * m_nVertexStride])) return -1;

        // Existing Data?
        if ( m_pVertex )
        {
            // Copy old data into new buffer
            memcpy( pVertexBuffer, m_pVertex, m_nVertexCount * m_nVertexStride );

            // Release old buffer
            delete []m_pVertex;

        } // End if

        // Store pointer for new buffer
        m_pVertex = pVertexBuffer;

    } // End if a resize is required
```

First we check to see whether the number of vertices passed in will exceed the current capacity of the vertex array. If so, then we need to resize the array. We grow the array by 100 vertices every time the capacity is reached so we will not need to resize again until we have added another 100 vertices.

Note the use of the loop rather than a simple conditional test for the resize. This handles the cases where a user adds large numbers of vertices in one go. If a user wanted to add 500 vertices, we would need to resize the array by 500, not 100. The loop incrementally adds 100 to the current m_nVertexCapcity variable until it is large enough to hold all the vertices required.

Next we use the local temporary pointer pVertexBuffer to allocate a new BYTE array large enough to hold the required number of vertices. Notice that we multiply the vertex capacity by the stride (m_nVertexStride) to get the total number of bytes needed. This is a clear example of why it is important to call SetDataFormat prior to adding vertex data.

If the vertex array currently contains vertex data, then we copy it into the newly allocated buffer and delete the original array because we no longer need it. Finally, we reassign the m_pVertex variable to point to the new vertex array which now contains any previous vertex data and enough room on the end to store the input vertex data. Next we copy over the vertex data passed into this function (it will be appended to the current contents of the buffer). Because we have not yet increased the internal vertex count variable, this tells us exactly where to start adding the new vertices.

```
    // Copy over vertex data if provided
    if ( pVertices )
       memcpy(&m_pVertex[m_nVertexCount*m_nVertexStride],pVertices,Count*m_nVertexStride);
```

Finally, we update the vertex count and return the index of the first newly added vertex in the array. This allows the calling function to retrieve a pointer to the vertex data and use it to start placing vertex data in the correct position.

```
    // Increase Vertex Count
    m_nVertexCount += Count;

    // Return first vertex
    return m_nVertexCount - Count;
}
```

CTriMesh::AddFace

The AddFace function is the second part of the geometry creation functionality. Since the ID3DXMesh object always represents its data as indexed triangle lists, we will follow the same rules when adding mesh indices.

The AddFace function is similar to the AddVertex function in the sense that it is used to temporarily store face data until the ID3DXMesh is created. But there is one important difference -- in addition to the face data (the indices), we must also specify an attribute ID describing the subset the face belongs to. These attribute IDs will be arbitrary values that are used by the application to group faces that have like properties. Therefore, the function must resize the temporary index array as well as the attribute array to allow for one attribute per index buffer triangle.

The first parameter is the number of triangles we wish to add and the second parameter is a void pointer to the index data for the triangles. This buffer should contain Count * 3 indices. The final parameter is the attribute ID we would like associated with the face(s) we are adding. This means that we can add multiple triangles with the same attribute ID in a single call. If we add N faces to the mesh, we will also need to add N attribute ID’s to the temporary pAttribute buffer.

```
long CTriMesh::AddFace( ULONG Count, LPVOID pIndices , ULONG AttribID  )
{
    UCHAR * pIndexBuffer = NULL;
    ULONG * pAttributeBuffer = NULL;
```

First we allocate two pointers that can be used to resize the index and attribute arrays. Because the index buffer contains faces and the attribute array holds an attribute ID for each face, these two arrays will always be resized together so that they stay in sync.

Resizing the arrays is identical to the vertex array resize. The default capacity and resize values are both 100.

```
    if ( m_nFaceCount + Count > m_nFaceCapacity )
    {
        // Adjust our face capacity (resize 100 at a time)
        for ( ; m_nFaceCapacity < (m_nFaceCount + Count) ; ) m_nFaceCapacity += 100;
```

We use the local pIndexBuffer pointer to allocate enough memory to hold the required indices. We multiply the desired face count by three to get the desired index count, and then multiply that value by the size of an index (2 or 4 bytes) to get the total number of bytes needed for the new temporary index buffer.

```
        // Allocate new resized array
        if (!( pIndexBuffer = new UCHAR[(m_nFaceCapacity * 3)*m_nIndexStride])) return -1;
```

If there is currently data in the index array then we will copy it into the new index array and release the old array. We reassign m_pIndex to the newly created array.

```
        // Existing Data?
        if ( m_pIndex )
        {
            // Copy old data into new buffer
            memcpy( pIndexBuffer, m_pIndex, (m_nFaceCount * 3) * m_nIndexStride );

            // Release old buffer
            delete []m_pIndex;
        }

        // Reassign pointer to new buffer
        m_pIndex = pIndexBuffer;
```

We do similar resizing to the attribute array.

```
        // Allocate new resized attribute array
        pAttributeBuffer = new ULONG[ m_nFaceCapacity ];
```

If the current attribute array (m_pAttribute) holds any data, then we copy it into the new array. Finally we release the old array and ressign the member variable m_pAttribute.

```
        // Existing Data?
        if ( m_pAttribute )
        {
            // Copy old data into new buffer
            memcpy( pAttributeBuffer, m_pAttribute, m_nFaceCount * sizeof(ULONG) );

            // Release old buffer
            delete []m_pAttribute;
        }

        // Store pointer for new buffer
        m_pAttribute = pAttributeBuffer;

    } // End if a resize is required
```

Next we append the index data passed into the function into the index array.

```
    // Copy over index and attribute data if provided
    if ( pIndices )
      memcpy(&m_pIndex[(m_nFaceCount*3)*m_nIndexStride],pIndices,(Count*3)*m_nIndexStride);
```

Finally, we loop through the new elements added to the attribute array (one for each new face) and set the attribute ID to the value passed into the function. We then increment the mesh face count and return the index of the first newly added face.

```
    for ( ULONG i = m_nFaceCount; i < m_nFaceCount + Count; ++i) m_pAttribute[i] = AttribID;

    // Increase Face Count
    m_nFaceCount += Count;

    // Return first face
    return m_nFaceCount - Count;
}
```

CTriMesh::AddAttributeData

As discussed earlier, a managed mesh maintains an array of MESH_ATTRIB_DATA structures describing the texture and material used by each subset. This allows the mesh to render itself. We use the CTriMesh::AddAttributeData function to add MESH_ATTRIB_DATA information to this array. This function is necessary if we manually create a mesh which is intended for use in managed mode. It allows us to specify the texture and material properties used by each subset to manually build the mesh's internal attribute array. This function serves no purpose for non-managed mode meshes.

The function takes one parameter which describes how many MESH_ATTRIB_DATA structures to make room for in the m_pAttribData array. If the array already contains mesh attribute data then the array will be resized to store the requested number of elements plus the elements already in the array. Array resizing works in exactly the same way as we saw in the last two functions discussed.

```
long CTriMesh::AddAttributeData( ULONG Count  )
{
    MESH_ATTRIB_DATA * pAttribBuffer = NULL;

    // Allocate new resized array
     pAttribBuffer = new MESH_ATTRIB_DATA[ m_nAttribCount + Count ] ;

    // Existing Data?
    if ( m_pAttribData )
    {
        // Copy old data into new buffer
        memcpy( pAttribBuffer, m_pAttribData, m_nAttribCount * sizeof(MESH_ATTRIB_DATA) );

        // Release old buffer
        delete []m_pAttribData;
    }
```

```
    // Store pointer for new buffer
    m_pAttribData = pAttribBuffer;

    // Clear the new items
    ZeroMemory( &m_pAttribData[m_nAttribCount], Count * sizeof(MESH_ATTRIB_DATA) );

    // Increase Attrib Count
    m_nAttribCount += Count;

    // Return first Attrib
    return m_nAttribCount - Count;
}
```

The function returns the index of the first attribute added by the call. The calling function can use the returned value to index into the m_pAttribData function and fill out the required attribute information. Keep in mind that the number of mesh attribute data elements in this array should match the number of subsets in the managed mesh. There is a one-to-one mapping between subsets in the mesh and this array, therefore, the first element you add to this array will be used to describe the texture and material information for the first subset, and so on.

One thing is not immediately clear. The above function allows us to add space in the internal attribute array of the mesh and returns the index of the newly added attribute element. How do we use this to place texture and material information into that array element? The mesh object also exposes a function called GetAttributeData which returns a pointer to the underlying attribute array. You can use this pointer along with the index returned from the above function to access the newly added attribute elements and populate them with texture and material data.

> **Note:** You should not call the AddAttributeData function if you intend to use the mesh in non-managed mode, since this array will only be used in managed mode. In non-managed mode, the application or scene class is responsible for setting a subset's texture and material before rendering it.

Example: Creating a cube using CTriMesh

We have now covered the basic functions we need to add data to the mesh. For illustration, the following code snippet will demonstrate how we might create a simple cube mesh using these functions. This next section is like an instruction manual for how to use the wrapper (as opposed to an examination of the code), and should be helpful in providing insight into the manual creation process.
First the application must allocate a new CTriMesh. In this example we will be using vertices with a 3D position and one set of 2D texture coordinates. After the mesh is created, the SetDataFormat function is called to inform the mesh object of the vertex and index formats intended for use:

```
CTriMesh  CubeMesh;
CubeMesh.SetDataFormat(  D3DFVF_XYZ | D3DFVF_TEX1 , 2 );
```

Because we will be adding one quad at a time in this example (a cube face is two triangles), we will use temporary arrays of 4 vertices and 6 indices to store the information for each face. We will pass this data into the CTriMesh::AddVertex and CTriMesh::AddFace functions respectively:

```
USHORT   Indices[6];
CVertex  Vertices[4];

// Build Front quad ( all quads point inwards in this example)
Vertices[0] = CVertex( -10.0f,  10.0f,  10.0f, 0.0f, 0.0f );
Vertices[1] = CVertex(  10.0f,  10.0f,  10.0f, 1.0f, 0.0f );
Vertices[2] = CVertex(  10.0f, -10.0f,  10.0f, 1.0f, 1.0f );
Vertices[3] = CVertex( -10.0f, -10.0f,  10.0f,, 0.0f, 1.0f );

// Build the front face indices
Indices[0] = 0; Indices[1] = 1; Indices[2] = 3;
Indices[3] = 1; Indices[4] = 2; Indices[5] = 3;
```

We assign the two triangles of this face to attribute 0 using the AddFace function:

```
// Add the vertices and indices to this mesh for front face
CubeMesh.AddVertex( 4, &Vertices );
CubeMesh.AddFace( 2, Indices, 0 );
```

We have now added the indices and the vertices to the mesh. The two triangles we have added belong to subset 0 and describe the front quad of the cube. Now we can re-use our local arrays to store the information for the back quad before adding it to the mesh. These vertices will be added to the mesh as vertices 4 through 7, so we must set the indices with this in mind.

```
// Back Quad
Vertices[0] = CVertex(  10.0f,  10.0f, -10.0f, 0.0f, 0.0f );
Vertices[1] = CVertex( -10.0f,  10.0f, -10.0f, 1.0f, 0.0f );
Vertices[2] = CVertex( -10.0f, -10.0f, -10.0f, 1.0f, 1.0f );
Vertices[3] = CVertex(  10.0f, -10.0f, -10.0f, 0.0f, 1.0f );

// Build the back quad indices
Indices[0] = 4; Indices[1] = 5; Indices[2] = 7;
Indices[3] = 5; Indices[4] = 6; Indices[5] = 7;

// Add the vertices and indices to this mesh
CubeMesh.AddVertex( 4, &Vertices );
CubeMesh.AddFace( 2, Indices, 0 );
```

This face has also been assigned an attribute ID of 0. Thus the two front triangles and the two back triangles of the cube belong to subset 0. As a result, these four triangles will be rendered with the same texture/material combination.

We use the same technique to add the left and right quads. Both will be assigned attribute IDs of 1 so they will belong to the same subset and be rendered with the same texture/material.

```
// Left Quad
Vertices[0] = CVertex( -10.0f,  10.0f, -10.0f, 0.0f, 0.0f );
Vertices[1] = CVertex( -10.0f,  10.0f,  10.0f, 1.0f, 0.0f );
Vertices[2] = CVertex( -10.0f, -10.0f,  10.0f, 1.0f, 1.0f );
Vertices[3] = CVertex( -10.0f, -10.0f, -10.0f, 0.0f, 1.0f );

// Build the left quad indices
Indices[0] = 8; Indices[1] = 9; Indices[2] = 11;
Indices[3] = 9; Indices[4] = 10; Indices[5] = 11;

// Add the vertices and indices to this mesh
CubeMesh.AddVertex( 4, &Vertices );
CubeMesh.AddFace( 2, Indices, 1 );

// Right Quad
Vertices[0] = CVertex(  10.0f,  10.0f,  10.0f, 0.0f, 0.0f );
Vertices[1] = CVertex(  10.0f,  10.0f, -10.0f, 1.0f, 0.0f );
Vertices[2] = CVertex(  10.0f, -10.0f, -10.0f, 1.0f, 1.0f );
Vertices[3] = CVertex(  10.0f, -10.0f,  10.0f,  0.0f, 1.0f );

// Build the right quad indices
Indices[0] = 12; Indices[1] = 13; Indices[2] = 15;
Indices[3] = 13; Indices[4] = 14; Indices[5] = 15;

// Add the vertices and indices to this mesh
CubeMesh.AddVertex( 4, &Vertices );
CubeMesh.AddFace( 2, Indices, 1 );
```

Finally we add the top and bottom quads as subset 2. It should be noted that while we have added the faces to the mesh in subset order, this is not a requirement. You will usually optimize the mesh once it is created so that faces are internally batched by subset anyway.

```
// Top Quad
Vertices[0] = CVertex( -10.0f,  10.0f, -10.0f, 0.0f, 0.0f );
Vertices[1] = CVertex(  10.0f,  10.0f, -10.0f, 1.0f, 0.0f );
Vertices[2] = CVertex(  10.0f,  10.0f,  10.0f, 1.0f, 1.0f );
Vertices[3] = CVertex( -10.0f,  10.0f,  10.0f,  0.0f, 1.0f );

// Build the indices
Indices[0] = 16; Indices[1] = 17; Indices[2] = 19;
Indices[3] = 17; Indices[4] = 18; Indices[5] = 19;

// Add the vertices and indices to this mesh
CubeMesh.AddVertex( 4, &Vertices );
CubeMesh.AddFace( 2, Indices, 2 );

// Bottom Quad
Vertices[0] = CVertex( -10.0f, -10.0f,  10.0f, 0.0f, 0.0f );
Vertices[1] = CVertex(  10.0f, -10.0f,  10.0f, 1.0f, 0.0f );
Vertices[2] = CVertex(  10.0f, -10.0f, -10.0f, 1.0f, 1.0f );
Vertices[3] = CVertex( -10.0f, -10.0f, -10.0f, 0.0f, 1.0f );
```

```
// Build the indices
Indices[0] = 20; Indices[1] = 21; Indices[2] = 23;
Indices[3] = 21; Indices[4] = 22; Indices[5] = 23;

// Add the vertices and indices to this mesh
CubeMesh.AddVertex( 4, &Vertices );
CubeMesh.AddFace( 2, Indices, 2 );
```

At this point the mesh has all vertices stored in its temporary vertex array, all indices in its temporary index array, and the attribute IDs for each face are stored in its temporary attribute array (describing three subsets).

If we were intending to use this mesh as a non-managed mesh then our work would be done (except for loading textures and making sure that they are set before rendering the correct subsets). However, in this example we will assume that this is a managed mesh with attribute IDs in the range of 0 to 2 (i.e., three subsets). Therefore, we need to add the attribute data to the mesh for each subset so that the mesh knows which textures and materials to set when rendering its subsets.

First we call the AddAttributeData function with a value of 3 so that the mesh allocates its MESH_ATTRIB_DATA array to hold 3 elements.

```
// Add the attribute data (We'll let the mesh manage itself)
CubeMesh.AddAttributeData( 3 );
```

Next we call CTriMesh::GetAttributeData which will return a pointer to the internal MESH_ATTRIB_DATA array so we can populate it with meaningful attribute data for each subset.

```
MESH_ATTRIB_DATA *pAttribData;
pAttribData = CubeMesh.GetAttributeData();
```

Now we can loop through each element and store the texture pointer and the material for the subset mapped to that attribute. In this example we are not storing normals in our vertices so will not be using the DirectX lighting pipeline. Therefore, we do not bother setting the material for each subset because it will not be used. We just create a default material and copy it into each attribute.

We also assume that pTexture is an application-owned array containing the three textures used by the subsets. These textures would have been created prior to this code being executed. What is worthy of note is that the texture and attribute callback functions are only used by the mesh when parsing X files. When the application manually creates meshes in this way, it is responsible for creating the required textures and storing their pointers in the mesh attribute array (in managed mode only).

```
D3DMATERIAL9 Material;
ZeroMemory( &Material, sizeof(D3DMATERIAL9));

// Set the attribute data for each subset
for ( ULONG i = 0; i < 3; ++i )
{
    pAttribData[i].Texture  = pTexture[I];
```

```
        pTexture[I]->AddRef();
        pAttribData[i].Material = Material;
    }
```

A good example of mesh creation happens when loading an IWF file. The IWF file contains all the texture filenames which would be initially created and stored by the scene. The IWF file also contains the face data and the material and texture mappings for each face. Using this information, the faces can be added to the mesh one triangle at a time and the attributes can be grouped and added as the subsets of the mesh.

All that is left to do at this point is instruct the CTriMesh object to build its internal ID3DXMesh object using the data we have added to the temporary arrays. Notice that when we call BuildMesh, we pass in the D3DXMESH_MANAGED member of the D3DXMESH enumerated type in this example. This indicates that we would like the mesh vertex and index buffers created in the managed resource pool so that they can support automatic recovery from a lost/reset device.

```
// Build the mesh
CubeMesh.BuildMesh( D3DXMESH_MANAGED, m_pD3DDevice );
```

CTriMesh::BuildMesh

The BuildMesh function should be called after the application has finished manually populating the mesh with data. It should not be called if we have loaded the data from an X file as the underlying mesh will have already been created. BuildMesh will generate the underlying ID3DXMesh object and fill its vertex and vertex buffers with the information currently stored in the temporary arrays. The code is shown next a few lines at a time. This listing has had some of the error checking removed to improve readability, but the error checking is performed in the actual source code.

The function takes three parameters. The first specifes zero or more members of the D3DXMESH enumerated type. This indicates which memory pool we wish the ID3DXMesh we are about to create to use for its vertex and index buffers. The second parameter is a pointer to the Direct3D device object which will own the mesh resources. The optional third parameter, which is set to TRUE by default, is a Boolean variable indicating whether we wish the temporary vertex, index and attribute arrays to be freed from memory after the ID3DXMesh has been successfully created.

Usually you would want to release these temporary storage bins as they are no longer required, but it can be useful to keep them around if the underlying mesh is to be created in the default pool. If the device should become lost, default pool meshes would need to be created again from scratch, requiring your application to manually add all the vertex and index data again. If you do not release the temporary storage bins, then a simple call to the CTriMesh::BuildMesh function is all that is needed to restore the mesh to its former status. The underlying ID3DXMesh will still be released and recreated, but the vertex and index data is still in the temporary bins and is automatically copied over by this function. This convenience obviously comes at the expense of increased memory requirement.

```
HRESULT CTriMesh::BuildMesh(ULONG Options, LPDIRECT3DDEVICE9 pDevice, bool ReleaseOriginals)
{
    HRESULT hRet;
    LPVOID  pVertices   = NULL;
    LPVOID  pIndices    = NULL;
    ULONG  *pAttributes = NULL;
```

If this CTriMesh object already has an ID3DXMesh assigned to it, we should release it. This allows us to call this function to rebuild meshes that have become invalid due to device loss / reset.

```
    // First release the original mesh if one exists
    if ( m_pMesh ) { m_pMesh->Release(); m_pMesh = NULL; }
```

Next we test the m_nIndexStride member variable to see if the user has set the index data format to 32-bit. By default D3DXCreateMeshFVF creates meshes with 16-bit indices, so we will need to modify the mesh creation options to include the D3DXMESH_32BIT flag in that case.

```
    // Force 32 bit mesh if required
    if ( m_nIndexStride == 4 ) Options |= D3DXMESH_32BIT;
```

Now we call D3DXCreateMeshFVF to create the ID3DXMesh. We pass in the number of faces and the number of vertices the mesh will require, followed by the creation options. We also pass the FVF flags that were registered with the mesh and a pointer to the device that will own the mesh. The final parameter is the CTriMesh::m_pMesh pointer that will point to a valid ID3DXMesh interface if the function is successful.

```
     // Create the blank empty mesh
    D3DXCreateMeshFVF(m_nFaceCount,m_nVertexCount,Options,m_nVertexFVF,pDevice,&m_pMesh);
```

At this point our D3DXMesh has been created, but it contains no data. The next step is to lock the mesh vertex, index, and attribute buffers and copy over all of the information stored in our temporary arrays.

```
    // Lock the vertex buffer and copy the data
    m_pMesh->LockVertexBuffer( 0, &pVertices );
    memcpy( pVertices, m_pVertex, m_nVertexCount * m_nVertexStride );
    m_pMesh->UnlockVertexBuffer();

    // Lock the index buffer and copy the data
    m_pMesh->LockIndexBuffer( 0, &pIndices );
    memcpy( pIndices, m_pIndex, (m_nFaceCount * 3) * m_nIndexStride );
    m_pMesh->UnlockIndexBuffer();

    // Lock the attribute buffer and copy the data
    m_pMesh->LockAttributeBuffer( 0, &pAttributes );
    memcpy( pAttributes, m_pAttribute, m_nFaceCount * sizeof(ULONG) );
    m_pMesh->UnlockAttributeBuffer();
```

Finally, if the ReleaseOriginals Boolean parameter was set to TRUE (the default setting) then the temporary storage bins are de-allocated and their pointers are set to NULL. We also set all other variables that describe the data in the temporary arrays to zero.

```
    // Release the original data if requested
    if ( ReleaseOriginals )
    {
        if ( m_pVertex    ) delete []m_pVertex;
        if ( m_pIndex     ) delete []m_pIndex;
        if ( m_pAttribute ) delete []m_pAttribute;

        m_nVertexCount    = 0;
        m_nFaceCount      = 0;
        m_pAttribute      = NULL;
        m_pIndex          = NULL;
        m_pVertex         = NULL;
        m_nVertexCapacity = 0;
        m_nFaceCapacity   = 0;
     }

    // We're done
    return S_OK;
}
```

At this point the mesh has been fully created and populated and can be rendered using the DrawSubset function or the Draw function (if this is a managed mesh).

CTriMesh::Draw

CTriMesh::Draw automates the rendering of a managed mesh and should not be used to render non-managed meshes. A managed mode mesh maintains internal texture and material resources for subset rendering. As this is not the case for non-managed meshes, this function only works in managed mode and will immediately return if called for a non-managed mesh.

```
void CTriMesh::Draw(  )
{
    LPDIRECT3DDEVICE9 pD3DDevice = NULL;

    // If the mesh has not been created yet bail
    if (!m_pMesh ) return;

    // This function is invalid if there is no managed data
    if ( !m_pAttribData ) return;

    // Retrieve the Direct3D device
    m_pMesh->GetDevice( &pD3DDevice );

    // Set the attribute data
    pD3DDevice->SetFVF( GetFVF() );

    // Render the subsets
    for ( ULONG i = 0; i < m_nAttribCount; ++i )
    {
        pD3DDevice->SetMaterial( &m_pAttribData[i].Material );
        pD3DDevice->SetTexture( 0, m_pAttribData[i].Texture );
```

```
        m_pMesh->DrawSubset( i );

    } // Next attribute

    // Release the device
    pD3DDevice->Release();
}
```

After checking for managed mode rendering, the function retrieves the device from the ID3DXMesh. We then loop through each subset of the mesh, set its texture and material, and call ID3DXMesh::DrawSubset with the subset attribute ID. Finally, we release the device interface because ID3DXMesh::GetDevice increments the device reference count before returning the pointer. CTriMesh::GetFVF is a simple wrapper that passes the request through to ID3DXMesh::GetFVF to get the FVF flags the mesh was created with.

CTriMesh::DrawSubset

DrawSubset provides an interface for rendering individual mesh subsets. It can be used in both managed and non-managed modes. It is in fact our only means for rendering a non-managed mesh. A higher level process will be required to set the texture, material, and other states for each subset prior to the render call (as seen in the CTriMesh::Draw function shown above). This allows the scene to batch render subsets from multiple meshes to achieve attribute order rendering across mesh boundaries. This function can also be called to render a subset of a managed mesh, in which case the texture and material of the subset will be set automatically.

```
void CTriMesh::DrawSubset( ULONG AttributeID )
{
    LPDIRECT3DDEVICE9 pD3DDevice = NULL;

    // Set the attribute data if managed mode mesh
    if ( m_pAttribData && AttributeID < m_nAttribCount )
    {
        // Retrieve the Direct3D device
        m_pMesh->GetDevice( &pD3DDevice );

        pD3DDevice->SetMaterial( &m_pAttribData[AttributeID].Material );
        pD3DDevice->SetTexture( 0, m_pAttribData[AttributeID].Texture );

        // Release the device
        pD3DDevice->Release();
    }

    //draw the subset
    m_pMesh->DrawSubset( AttributeID );
}
```

If the mesh attribute data array is defined, then this is a managed mesh and we set the subset texture and material before rendering. If it is non-managed, then we simply call ID3DXMesh::DrawSubset straight away.

CTriMesh::OptimizeInPlace

CTriMesh::OptimizeInPlace is basically a wrapper around the ID3DXMesh::OptimizeInPlace function.

The first parameter is a DWORD containing one or more D3DXMESHOPT flags describing the optimization we wish to perform. The second and third parameters can be set to NULL or can be passed the address of ID3DXBuffer interface pointers that will contain the face and vertex remap information respectively. These should not be allocated buffers since the function will create both buffers for you.

```
HRESULT CTriMesh::OptimizeInPlace( DWORD Flags, LPD3DXBUFFER *ppFaceRemap,
                                   LPD3DXBUFFER *ppVertexRemap)
{
    HRESULT      hRet;
    LPD3DXBUFFER pFaceRemapBuffer = NULL;
    ULONG       *pData = NULL;
```

If the mesh has not yet had its adjacency information generated then we do so at this point. The m_pAdjacency member variable is a pointer to an ID3DXBuffer interface. The GenerateAdjacency function will create the D3DXBuffer object and fill it with adjacency information.

```
    // Generate adjacency if none yet provided
    if (!m_pAdjacency)
    {
        GenerateAdjacency();
    }
```

ID3DXMesh::OptimizeInPlace can be somewhat confusing when it comes to generating remap information. To get the face remap information you pass in a pointer to a pre-allocated DWORD array. To get vertex remap information you just pass a pointer to an ID3DXBuffer and the function will generate and fill it with the relevant data. To avoid confusion, our function will return both face and vertex remapping information in ID3DXBuffer objects. Therefore, we must first create the face remap buffer ourselves, obtain a pointer to its data area, cast it to a DWORD, and pass it into ID3DXMesh::OptimimzeInPlace.

```
    // Allocate the output face remap if requested
    if ( ppFaceRemap )
    {
        D3DXCreateBuffer( GetNumFaces() * sizeof(ULONG), ppFaceRemap );
         pData = (ULONG*)(*ppFaceRemap)->GetBufferPointer();
    }
```

Finally, we call ID3DXMesh::OptimizeInPlace to optimize the mesh data.

```
    // Optimize the data
    m_pMesh->OptimizeInplace(Flags, (DWORD*)m_pAdjacency->GetBufferPointer(),
                             (DWORD*)m_pAdjacency->GetBufferPointer(), pData, ppVertexRemap );
    return S_OK;
}
```

CTriMesh::Optimize

This function is a bit more complex than its predecessor due to the fact that it has to clone the optimized data into a new CTriMesh object. However, unlike ID3DXMesh::Optimize which automatically creates the output mesh, CTriMesh::Optimize expects the application to pass in a pre-created CTriMesh object. This mesh will be the recipient of the optimized data. This affords us the extra flexibility of being able to use a statically allocated or stack allocated CTriMesh object as the output mesh. This might be useful if you wanted to create a temporary optimized mesh locally inside a function, where the output mesh would be allocated on the stack and automatically discarded when the function returns.

The first parameter is a combination of D3DXMESH flags describing how the underlying ID3DXMesh object in the output mesh should be created. This can be combined with one or more D3DXMESHOPT flags describing the optimization to perform. The second parameter is a pointer to a CTriMesh object that will receive the optimized ID3DXMesh object. This CTriMesh pointer should point to an already created CTriMesh object. If the output mesh already contains an underlying ID3DXMesh object, this mesh will be released in favor of the new one that is created by this function.

```
HRESULT CTriMesh::Optimize(ULONG Flags, CTriMesh *pMeshOut, LPD3DXBUFFER *ppFaceRemap,
                           LPD3DXBUFFER *ppVertexRemap , LPDIRECT3DDEVICE9 pD3DDevice  )
{
    HRESULT             hRet;
    LPD3DXMESH          pOptimizeMesh = NULL;
    LPD3DXBUFFER        pFaceRemapBuffer = NULL;
    LPD3DXBUFFER        pAdjacency = NULL;
    ULONG              *pData      = NULL;
```

If the mesh we are cloning does not yet have its adjacency information generated, then we need to generate it, because it is needed in the call to ID3DXMesh::Optimize.

```
    // Generate adjacency if not yet available
    if (!m_pAdjacency)
    {
        hRet = GenerateAdjacency();
    }
```

If the caller did not pass in a pointer to a device, then we will use the device of the current mesh being cloned. Typically you will not want to specify a different device unless you are using multiple devices (very rare). Keep in mind that while the CTriMesh object is not bound to any particular device, its underlying ID3DXMesh object is. The next section of code calls CTriMesh::GetDevice to get a pointer to the current mesh's device if a device pointer was not specified. If a device pointer was passed, then we increment its reference count until we finish using it.

```
    if ( !pD3DDevice )
    {
        // we'll use the same device as this mesh
        // This automatically calls 'AddRef' for the device
        m_pMesh->GetDevice( &pD3DDevice );
```

```
    }
    else
    {
        // Otherwise we'll add a reference here so that we can
        // release later for both cases without doing damage :)
        pD3DDevice->AddRef();
    }
```

ID3DXMesh::Optimize also returns face adjacency information for the optimized mesh, so we create a buffer to store this information. As with the OptimizeInPlace method, we create the face remap buffer if a face remap pointer was passed into the function.

```
    // Allocate new adjacency output buffer
    D3DXCreateBuffer( (3 * GetNumFaces()) * sizeof(ULONG), &pAdjacency );

    // Allocate the output face remap if requested
    if ( ppFaceRemap )
    {
        // Allocate new face remap output buffer
        D3DXCreateBuffer( GetNumFaces() * sizeof(ULONG), ppFaceRemap );
        pData = (ULONG*)(*ppFaceRemap)->GetBufferPointer();
   }
```

At the top of this function we declared a local ID3DXMesh pointer called pOptimizeMesh. This pointer will be fed into the ID3DXMesh::Optimize function and will point to the newly created optimized mesh on function return.

```
   // Attempt to optimize the mesh
   m_pMesh->Optimize( Flags,
                      (ULONG*)m_pAdjacency->GetBufferPointer(),
                      (ULONG*)pAdjacency->GetBufferPointer(),
                       pData, ppVertexRemap, &pOptimizeMesh );
```

At this point we now have an optimized ID3DXMesh, but it is not yet connected to the output CTriMesh object that was passed into this function. So we call the CTriMesh::Attach method for the output CTriMesh object which points the m_pMesh pointer to the optimized ID3DXMesh. Notice that we also pass the face adjacency buffer so that the new mesh has this data resident.

```
    // Attach this D3DX mesh to the output CTriMesh
    // This automatically adds a reference to the mesh passed in.
    pMeshOut->Attach( pOptimizeMesh, pAdjacency );
```

The optimized mesh and its adjacency buffer have now been assigned to the output CTriMesh, which increases the reference count on both. We no longer need to use these interfaces in this function so we release them. This does not destroy the buffer or the mesh because they are now being referenced by the output CTriMesh object.

```
    // We can now release our copy of the optimized mesh and the adjacency buffer
    pOptimizeMesh->Release();
    pAdjacency->Release();
```

If we are optimizing a managed mesh, then the source mesh will have an array of mesh attribute data elements describing the textures and materials used by each subset. If this is the case, we must also copy this data into the mesh attribute array of the output CTriMesh. First we call AddAttributeData to make room for the new attributes and then we copy the data over.

```
    // Copy over attributes if there is anything here
    if ( m_pAttribData )
    {
        // Add the correct number of attributes
         pMeshOut->AddAttributeData( m_nAttribCount ) ;

        // Copy over attribute data
        MESH_ATTRIB_DATA * pAttributes = pMeshOut->GetAttributeData();
        for ( ULONG i = 0; i < m_nAttribCount; ++i )
        {
            MESH_ATTRIB_DATA * pAttrib = &pAttributes[i];

            // Store details
            pAttrib->Material = m_pAttribData[i].Material;
            pAttrib->Texture  = m_pAttribData[i].Texture;
            pAttrib->Effect   = m_pAttribData[i].Effect;

            // Add references so that objects aren't released when either of these
            // meshes are released, or vice versa.
            if ( pAttrib->Texture ) pAttrib->Texture->AddRef();
            if ( pAttrib->Effect  ) pAttrib->Effect->AddRef();
        } // Next Attribute
    } // End if managed

    // Release our referenced D3D Device
    if (pD3DDevice) pD3DDevice->Release();

    // Success!!
    return S_OK;
}
```

Notice that when we copy over each texture pointer (and effect pointer) we make sure that we increase the reference count so that it correctly reflects how many external pointers to the interface are in existence.

CTriMesh::GenerateAdjacency

The CTriMesh class includes a member variable called m_pAdjacency which is a pointer to an ID3DXBuffer interface. This function creates this buffer and calls ID3DXMesh::GenerateAdjacency to calculate the adjacency information. ID3DXMesh::GenerateAdjacency expects a DWORD pointer to an array large enough to hold the adjacency information, so we simply allocate the ID3DXBuffer large enough to hold three DWORDS per face and retrieve the buffer data pointer and cast it before passing it into the function. CTriMesh::GenerateAdjacency accepts an optional epsilon parameter which will default to 0.001. This is used as a tolerance value when comparing vertices in neighboring faces to see if they are joined.

Note: This function should only be called after the underlying ID3DXMesh has been created.

```
HRESULT CTriMesh::GenerateAdjacency( float Epsilon )
{
    HRESULT hRet;

    // Validate Requirements
    if ( !m_pMesh ) return D3DERR_INVALIDCALL;

    // Clear out any old adjacency information
    if (m_pAdjacency) m_pAdjacency->Release();

    // Create the new adjacency buffer
    hRet = D3DXCreateBuffer( GetNumFaces() * (3 * sizeof(DWORD)), &m_pAdjacency );
    if ( FAILED(hRet) ) return hRet;

    // Generate the new adjacency information
    hRet = m_pMesh->GenerateAdjacency( Epsilon, (DWORD*)m_pAdjacency->GetBufferPointer() );
    if ( FAILED(hRet) ) return hRet;

    // Success !!
    return S_OK;
}
```

CTriMesh::Attach

The Attach function is used when a new ID3DXMesh is created and needs to be attached to a pre-existing CTriMesh object. Its parameters are an ID3DXMesh and its adjacency buffer. The adjacency buffer passed is simply stored for later use. If you do not pass the adjacency buffer it will be generated when it is needed. The function returns NULL if a valid mesh pointer is not passed.

```
HRESULT CTriMesh::Attach( LPD3DXBASEMESH pMesh, LPD3DXBUFFER pAdjacency /* = NULL */ )
{
    HRESULT hRet;

    // Validate Requirements
    if ( !pMesh ) return D3DERR_INVALIDCALL;

    // Clear our current data
    Release();
```

We start by calling CTriMesh::Release to release any prior mesh or adjacency data. Since the input mesh is the generic base class, we use a method of the IUnknown interface to determine whether or not the passed pointer is to the correct COM interface (ID3DXMesh in this case). We call the IUnkown::QueryInterface method and pass the interface type we are querying and the m_pMesh pointer. If the mesh is a valid ID3DXMesh COM interface, it will be copied into the m_pMesh pointer and its reference count incremented automatically by the QueryInterface call.

```
    // Store this mesh (ensuring that it really is of the expected type)
```

```
    // This will automatically add a reference of the type required
    hRet = pMesh->QueryInterface( IID_ID3DXMesh, (void**)&m_pMesh );
    if ( FAILED(hRet) ) return hRet;
```

Next, we copy FVF, vertex and index stride, and the adjacency buffer (if it was passed in). We remember to call AddRef to ensure proper reference counting on the buffer. Once done, we return success.

```
    // Calculate strides etc
    m_nVertexFVF    = m_pMesh->GetFVF();
    m_nVertexStride = m_pMesh->GetNumBytesPerVertex();
    m_nIndexStride  = (GetOptions() & D3DXMESH_32BIT) ? 4 : 2;

    // If adjacency information was passed, reference it
    // if none was passed, it will be generated later, if required.
    if ( pAdjacency )
    {
        m_pAdjacency = pAdjacency;
        m_pAdjacency->AddRef();
    }
    // Success!!
    return S_OK;
}
```

CTriMesh::CloneFVF

A useful feature of the ID3DXMesh interface is the ability to clone the mesh into a new mesh with a different vertex/index format. This is especially handy when we load data from an X file which may lack certain pieces of information we need or more components than we intend to support.

This function is almost identical to our Optmize function, without the actual optimization call. The function simply clones and attaches the mesh based on the creation options and FVF specified as input parameters. Adjacency data is either copied or generated and attributes are copied if the mesh is in managed mode. The final mesh is passed out using the third parameter.

```
HRESULT CTriMesh::CloneMeshFVF( ULONG Options, ULONG FVF, CTriMesh * pMeshOut,
                                LPDIRECT3DDEVICE9 pD3DDevice /* = NULL */ )
{
    HRESULT             hRet;
    LPD3DXMESH          pCloneMesh  = NULL;

    // Validate requirements
    if ( !m_pMesh || !pMeshOut ) return D3DERR_INVALIDCALL;

    // Generate adjacency if not yet available
    if (!m_pAdjacency)
    {
       GenerateAdjacency();
    }

    // If no new device was passed...
    if ( !pD3DDevice )
```

```
    {
        // we'll use the same device as this mesh
        // This automatically calls 'AddRef' for the device
        m_pMesh->GetDevice( &pD3DDevice );
    }
    else
    {
        // Otherwise we'll add a reference here so that we can
        // release later for both cases without doing damage :)
        pD3DDevice->AddRef();
    }

    // Attempt to clone the mesh
    m_pMesh->CloneMeshFVF( Options, FVF, pD3DDevice, &pCloneMesh );

    // Attach this D3DX mesh to the output mesh
    // This automatically adds a reference to the mesh passed in.
    pMeshOut->Attach( pCloneMesh );

    // We can now release our copy of the cloned mesh
    pCloneMesh->Release();

    // Copy over attributes if there is anything here
    if ( m_pAttribData )
    {
        // Add the correct number of attributes
        if ( pMeshOut->AddAttributeData( m_nAttribCount ) < 0 ) return E_OUTOFMEMORY;

        // Copy over attribute data
        MESH_ATTRIB_DATA * pAttributes = pMeshOut->GetAttributeData();
        for ( ULONG i = 0; i < m_nAttribCount; ++i )
        {
            MESH_ATTRIB_DATA * pAttrib = &pAttributes[i];
            // Store details
            pAttrib->Material = m_pAttribData[i].Material;
            pAttrib->Texture  = m_pAttribData[i].Texture;
            pAttrib->Effect   = m_pAttribData[i].Effect;

            // Add references so that objects aren't released when either of these
            // meshes are released, or vice versa.
            if ( pAttrib->Texture ) pAttrib->Texture->AddRef();
            if ( pAttrib->Effect  ) pAttrib->Effect->AddRef();

        } // Next Attribute

    } // End if managed
    // Release our referenced D3D Device
    if (pD3DDevice) pD3DDevice->Release();

    // Success!!
    return S_OK;
}
```

CTriMesh::WeldVertices

WeldVertices is a thin wrapper around the D3DXWeldVertices function.

```
HRESULT CTriMesh::WeldVertices( ULONG Flags,  const D3DXWELDEPSILONS * pEpsilon  )
{
    HRESULT          hRet;
    D3DXWELDEPSILONS WeldEpsilons;

    // Validate Requirements
    if ( !m_pMesh ) return D3DERR_INVALIDCALL;

    // Generate adjacency if none yet provided
    if (!m_pAdjacency)
    {
        GenerateAdjacency();
    } // End if no adjacency

    // Fill out an epsilon structure if none provided
    if ( !pEpsilon )
    {
        // Set all epsilons to 0.001;
        float * pFloats = (float*)&WeldEpsilons;
        for ( ULONG i = 0; i < sizeof(D3DXWELDEPSILONS) / sizeof(float); i++ )
            *pFloats++ = 1e-3f;

        // Store a pointer (this doesn't get passed back or anything,
        // we're just reusing the empty var)
        pEpsilon = &WeldEpsilons;

    } // End if

    // Weld the vertices
    D3DXWeldVertices( m_pMesh, Flags, pEpsilon, (DWORD*)m_pAdjacency->GetBufferPointer(),
                      (DWORD*)m_pAdjacency->GetBufferPointer(), NULL, NULL );
     // Success!!
    return S_OK;
}
```

This function takes two parameters. The first is a flag containing zero or more members of the D3DXMESH enumerated type. While these are usually used as mesh creation flags and this function does not create a new output mesh, the weld operation will regenerate the vertex and index buffers. This means that we can optionally use the weld function to change the resource pool allocation strategy while we are welding it.

The second parameter is a pointer to a D3DXWELDEPSILONS structure which allows us to specify floating point tolerances for each possible vertex component. Specifying NULL for the second parameter will force the function to use a default comparison tolerance of 0.001 for each vertex component.

The D3DXWeldVertices function needs to be passed the face adjacency information, so if our CTriMesh object has not yet generated it, we tell it to do so. Next we test to see if a valid pointer to a

D3DXWELDEPSILONS structure was passed. If not, then we temporarily create one and set all of its tolerance values to 0.001. Finally, we call D3DXWeldVertices to perform the weld on the ID3DXMesh for this CTriMesh object.

CTriMesh::GetNumFaces

There are a number of CTriMesh functions we will not discuss here. They are all GetXX style functions which are generally one or two lines long. However we will use this one method as an example of the rest.

```
ULONG CTriMesh::GetNumFaces( ) const
{
    // Validation!!
    if ( !m_pMesh )
        return m_nFaceCount;           // Number of faces in temp index buffer
    else
        return m_pMesh->GetNumFaces(); // Number of faces in actual ID3DXMesh
}
```

This is a good example of how all of the GetXX functions work. Since the user may be creating meshes manually, they might call this function before calling the Build function to create the ID3DXMesh object. In this case, the function should return the number of faces that have currently been added to the temporary index arrays (m_nFaceCount). If the ID3DXMesh has been created however, then we call the ID3DXMesh::GetNumFaces function to return the number of faces in the actual mesh object. Many of these GetXX functions are built to work in these two modes.

Introducing IWF External References

In Chapter Five we discussed loading IWF files using the IWF SDK. While most of that code will remain intact, in this project we will be introducing some new IWF chunks that are specific to GILES™.

GILES™ v1.5 and above includes a new data type called a *reference*. References allow the artist to place multiple objects in the scene which all share the same physical mesh data. This is very much like the concept of instances we discussed in Chapter One. We only need to store one copy of the vertices and indices in memory and then have multiple objects reference the data. Any changes to the data will affect all objects that are referencing it. References can point to internal objects such as those created in GILES™ or they can point to external files. In the latter case, the IWF will contain filenames for X files. The X file data is not included within the IWF file, only the name is stored. Thus when we find an external reference, we will load the file ourselves.

External reference objects are very useful because they allow us to use GILES™ to place and position objects in the scene while still keeping the mesh data separate from the scene database. This means that if we need to tweak the polygons in one of our X files, we can do so without having to resave the scene. This also means that we can use GILES™ to create the scene and position placeholder objects in the world even before the artist has finalized the assets. Once the model is complete, it can be dropped into the game folder and (as long as the name has not changed) our code will display the scene properly.

You will recall from earlier lessons how the IWF SDK provides us with a helper function to automate the loading of IWF files. After the file has been loaded, the various components of the scene are stored in lists. As a simple example, all faces are stored in an IWFSurface list, all entities are stored in IWFEntity lists, etc. GILES™ reference objects (internal and external) are implemented as an entity plugin. This means that they will be stored in the entity list created by the IWF SDK loader, not the mesh list, and will require our code to load the mesh data using the given filename or object name contained inside the entity.

Below you can see how the reference entity is laid out in memory inside the entity's data area. This is the structure we will use in our CScene class to access the data stored in a reference entity loaded by the IWF SDK.

```
typedef struct _ReferenceEntity
{
    ULONG       ReferenceType;          // What type of reference is this
    ULONG       Flags;                  // Reserved flags
    ULONG       Reserved1;              // Reserved values
    ULONG       Reserved2;              // Reserved values
    ULONG       Reserved3;              // Reserved values
    char        ReferenceName[1024];    // External file name or internal object name
};
```

Many of these members are reserved for future use and can be ignored. Our focus is on the ReferenceType (0 for internal references, 1 for external references) and the ReferenceName (filename for external refernces, object name for internal references). For internal references, the name will be the

name of another object stored inside the IWF file. In GILES™, we can give objects in the scene unique names so that they can be referenced in this way. To keep things simple, our first application will support only external references (references to X files). This allows us to demonstrate how to use CTriMesh::LoadMeshFromX to load mesh data for each object in the scene stored in an IWF file.

> **Note:** The IWF file distributed with this project (Space_Scene.iwf) is unlike others we have loaded in past lessons. This file contains no actual geometry data. It contains only a skybox and a list of external reference entities. The scene uses four meshes, each stored in its own X file. Three of the meshes are spacecraft models (Cruiser1.x, Cruiser2.x, sfb-04.x). These objects were positioned in the scene using external referencing within GILES, so that only the object world matrix is stored in the IWF file along with its filename. The fourth X file is called Asteroid.x and is referenced many times in the scene. When we load these objects, we will give each reference its own CObject structure and attach the appropriate CTriMesh to each. From that point forward, rendering proceeds as usual.

The CScene Class

Our scene class will have several responsibilities:

- File loading
- Resource management
  - Textures
  - Materials
  - (Attributes)
  - Meshes
- Rendering

For mesh data, the scene class must be prepared to:

- Load externally referenced X files contained in IWF files
- Manually fill meshes with IWF data
- Manually fill meshes with procedural data (i.e. skyboxes, cubes, etc.)

CGameApp::BuildObjects starts asset loading with a call to CScene::LoadSceneFromIWF. This is very similar to previous projects we have developed. As before, we will use the CFileIWF class that ships with the IWF SDK to handle file import automatically on our behalf. This object contains a number of STL vectors for storage of entities, meshes, texture names, materials, etc that were extracted from the file. We will implement a number of processing functions (ProcessMeshes, ProcessEntities, etc.) to extract the relevant data from these vectors and convert them into a format which our scene object accepts.

> **Note:** We will not be covering all of the code in the CScene class since much of it is unchanged from previous applications. There are some small alterations where we might now copy data into a CTriMesh object instead of into a CMesh structure, but that is about the extent of it. We will focus on the major areas that have undergone revision such as data processing and rendering.

An abbreviated version of the CScene class declaration follows. The full declaration can be found in the header file CScene.h.

```
class CScene
{
public:
    //-----------------------------------------------------------------------
    // Constructors & Destructors for This Class.
    //-----------------------------------------------------------------------
     CScene( );
    ~CScene( );

    //-----------------------------------------------------------------------
    // Public Functions for This Class
    //-----------------------------------------------------------------------
    bool                LoadSceneFromIWF( TCHAR * strFileName,
                                          ULONG LightLimit = 0,
                                          ULONG LightReservedCount = 0 );

    bool                LoadSceneFromX  ( TCHAR * strFileName );

    void                Render          ( CCamera & Camera );
    void                RenderSkyBox    ( CCamera & Camera );
```

LoadSceneFromIWF will be called from CGameApp at application startup. This function was available in previous lessons when we discussed light groups (not used in this demo). The LightLimit parameter in the LoadSceneFromIWF file will simply be used to indicate how many lights we wish to load from the IWF file. For example, if the current hardware only supports eight simultaneous lights, we would set this to 8 so that only the first eight lights in the IWF file would be used and the rest would be ignored. In this application, the third parameter is not used but you may remember that it was used in the light group demo to reserve a number of light slots for dynamic lights.

The CScene class also has a LoadSceneFromX function which provides loading of X files which may contain individual or multiple meshes. Unlike LoadSceneFromIWF which gives each mesh loaded its own world matrix, LoadSceneFromX creates a single CTriMesh for the entire scene. If the X file contains multiple meshes, they will be collapsed into a single mesh. We will learn in the next chapter how to load hierarchical X files containing multiple meshes. In that case each mesh will maintain its own world matrix and can be manipulated separately from other meshes stored in the same X file.

CScene also has two render functions: CScene::Render is called by CGameApp::FrameAdvance to draw all scene meshes. CScene::RenderSkyBox draws the skybox and is called prior to mesh rendering to present a nice backdrop for the scene.

The next two methods are static callback functions that the scene will register with the CTriMesh class to handle the loading of textures and materials. We need two callback functions to differentiate the behaviour of managed vs. non-managed mesh modes. A managed mesh will call the CollectTexture function to load textures and store their pointers in its attribute data array. Non-managed meshes call CollectAttributeID to search the scene database for a material/texture combo index which is used to re-map the mesh attribute buffer to reference the global resource list.

```
    //-----------------------------------------------------------------------
    // Static Public Functions for This Class
    //-----------------------------------------------------------------------
    static LPDIRECT3DTEXTURE9   CollectTexture ( LPVOID pContext, LPCTSTR FileName );
    static ULONG                CollectAttributeID (
                                        LPVOID pContext, LPCTSTR strTextureFile,
                                        const D3DMATERIAL9 * pMaterial,
                                        constD3DXEFFECTINSTANCE *pEffectInstance=NULL);
```

In our demo project, we will use a managed mesh for our skybox because it shares no resources with other objects in the scene and non-managed meshes for all scene geometry. This should give you a good sense of how these types can be used in the same application.

After LoadSceneFromIWF calls CFileIWF::Load, the IWF file data is stored in the CFileIWF object's internal vectors. As in previous demos, the scene will then extract the required data from those vectors using a series of ProcessXX functions.

```
    private:
    //-----------------------------------------------------------------------
    // Private Functions for This Class
    //-----------------------------------------------------------------------
    bool                ProcessMeshes     ( CFileIWF & pFile );
    bool                ProcessVertices   ( CTriMesh * pMesh, iwfSurface * pFilePoly );
    bool                ProcessIndices    ( CTriMesh * pMesh, iwfSurface * pFilePoly,
                                            ULONG AttribID = 0, bool BackFace = false);
    bool                ProcessMaterials  ( const CFileIWF& File );
    bool                ProcessTextures   ( const CFileIWF& File );
    bool                ProcessEntities   ( const CFileIWF& File );
    bool                ProcessReference  ( const ReferenceEntity& Reference,
                                            const D3DXMATRIX & mtxWorld );
    bool                ProcessSkyBox     ( const SkyBoxEntity& SkyBox );
    long                AddMesh           ( ULONG Count = 1 );
    long                AddObject         ( ULONG Count = 1 );
```

Note that there is an AddObject function and an AddMesh function which are used for adding CObject structures to the scene's CObject array and CTriMesh objects to the scene's CTriMesh array respectively.

The first new class member variable is a single statically allocated CTriMesh object that will be used to store a skybox. Our scene will contain one skybox (at most) and it will be manually created when necessary. GILES™ does not export geometry information for a skybox, only the six texture file names needed to create the effect. So when the scene class encounters a skybox entity in the IWF file, it will extract the texture file names, create the textures and then create the cube faces manually. The SkyBox mesh will be created in managed mode so that it will render in a self-contained manner.

```
    //-----------------------------------------------------------------------
    // Private Variables for This Class
    //-----------------------------------------------------------------------
    CTriMesh            m_SkyBoxMesh;               // Sky box mesh.
```

The scene class will also be responsible for resource allocation and management regardless of the mesh management mode selected. A callback mechanism is provided to allow scene loading functions access to texture resources when needed. In order to avoid duplicating texture resources, the scene will store the texture pointer along with its filename in the structure shown below.

```
typedef struct _TEXTURE_ITEM
{
       LPSTR               FileName;   // File used to create the texture
       LPDIRECT3DTEXTURE9  Texture;    // The texture pointer
} TEXTURE_ITEM;
```

The global resource pool for textures is stored in the CScene class as a single array:

```
    TEXTURE_ITEM       *m_pTextureList [MAX_TEXTURES];      // Array of texture pointers
```

In the case of a managed mesh, the scene will return a pointer to the texture back to the mesh loading function after it has added the texture to this array. The managed mesh can store the texture pointer in its attribute data array for later use. In the case of a non-managed mesh, the texture will still be added to this array by the callback function, but the pointer will not be returned to the mesh loader. Instead, the index of a texture/material combination will be returned and used to re-map the mesh attribute buffer.

A similar strategy to avoid duplication is used for material resources, but only for non-managed meshes. Managed meshes will store their own materials internally, although as an exercise, you should be able to quickly modify the class to reference the global material pool and reduce memory requirements. The scene material list will contain all materials used by non-managed meshes.

```
    D3DMATERIAL9        m_pMaterialList[MAX_MATERIALS];    // Array of material structures.
```

The scene class also uses a D3DLIGHT9 array to store the lights that were loaded from the IWF file. To simplify the code, this particular demo project does not use light groups, so only the first N lights are loaded from the file (where N is the total number of simultaneous lights supported by the device). You could certainly add light group support as an exercise however.

```
    D3DLIGHT9           m_pLightList   [MAX_LIGHTS];          // Array of light structures
```

The next three variables describe how many items are in the arrays just discussed.

```
    ULONG               m_nTextureCount;          // Number of textures stored
    ULONG               m_nMaterialCount;         // Number of materials stored
    ULONG               m_nLightCount;            // Number lights stored here
```

The space scene IWF file used in this project contains three space ships and a number of asteroids. The scene object will thus contain an array of CObject structures for each object in the scene that needs to be rendered. The CObject structure contains a world matrix and a pointer to a CTriMesh. Recall that multiple CObject's may reference the same CTriMesh. This is the approach used for our asteroids and is a technique referred to as mesh instancing. The scene also contains an array of CTriMesh objects for the

actual geometry data used by the scene objects. For example, the CTriMesh array will contains 4 meshes (3 space ship meshes and 1 asteroid mesh).

```
    CObject           **m_pObject;             // Array of objects storing meshes
    ULONG               m_nObjectCount;        // Number of objects currently stored
    CTriMesh          **m_pMesh;               // Array of loaded scene meshes
    ULONG               m_nMeshCount;          // Number of meshes currently stored
```

Non-managed meshes are not aware of the texture or material each of its subsets is using, since this information is managed at the scene level. The mesh object attribute buffer IDs will simply reference a texture and material combination stored in the CScene ATTRIBUTE_ITEM array shown next.

```
    ATTRIBUTE_ITEM      m_pAttribCombo[MAX_ATTRIBUTES]; // Table of attribute combinations
    ULONG               m_nAttribCount;                 // Number of attributes.
```

The ATTRIBUTE_ITEM structure represents a unique texture/material pair. This allows us to batch subsets across mesh boundaries when they share the same attribute combination (used by non-managed meshes only).

```
typedef struct _ATTRIBUTE_ITEM
{
    long TextureIndex;          // Index into the texture array
    long MaterialIndex;         // Index into the material array
} ATTRIBUTE_ITEM;
```

When a matching attribute item is found during the mesh loading callback function for non-managed meshes, an index into this global array is returned and stored in the mesh attribute buffer. If a match was not found, a new ATTRIBUTE_ITEM is created and inserted into the list and its index returned. The global texture and material arrays are used to avoid resource duplication as mentioned previously (note that the ATTRIBUTE_ITEM members are indices into the scene global texture and material arrays).

Finally, the scene stores a default material that will be used for any faces that do not have a material explicitly assigned in the file.

```
    D3DMATERIAL9        m_DefaultMaterial;   // A plain white material for null cases.
};
```

Now that we have examined the CScene member variables, let us look at the member functions. We will try to do this in the approximate order that they would be called so that we can better understand how the scene is built.

CScene::LoadSceneFromIWF

This function manages the loading of IWF files from disk. We pass in the filename of the IWF file we wish to load (with absolute or relative path) as well as the device light limit and a count for reserved light slots for dynamic lights. This project will not use light groups so we can ignore the third parameter.

The first thing the function does is instantiate a CFileIWF object provided us by the IWF SDK. We then strip off the filename portion of the input string and store the path portion in the m_strDataPath member variable. This path will be used later to load textures that are stored in the same folder.

```
bool CScene::LoadSceneFromIWF(TCHAR *strFileName, ULONG LightLimit, ULONG ReservedCount  )
{
    CFileIWF File;

    // Retrieve the data path
    if ( m_strDataPath ) free( m_strDataPath );
    m_strDataPath = _tcsdup( strFileName );

    // Strip off the filename
    TCHAR * LastSlash = _tcsrchr( m_strDataPath, _T('\\') );
    if (!LastSlash) LastSlash = _tcsrchr( m_strDataPath, _T('/') );
    if (LastSlash) LastSlash[1] = _T('\0'); else m_strDataPath[0] = _T('\0');
```

The next step is loading the IWF file from disk using the CFileIWF::Load function.

```
    // File loading may throw an exception
    try
    {
        // Attempt to load the file
        File.Load( strFileName );
```

At this point, all of the meshes, materials, texture names, and entities have been loaded and are stored in a series of STL vectors internal to the CFileIWF object. We now record the input maximum light count in a scene member variable.

```
        // Store values
        m_nLightLimit     = LightLimit;
```

The remainder of the function calls the ProcessXX member functions to extract the stored data into application defined data types. The first two calls (ProcessMaterials/ProcessTextures) extract the scene materials and textures and store them in their respective global arrays.

Interestingly, the space scene IWF file that accompanies this demo does not contain material or texture information because it does not store any actual mesh data. Instead, all meshes are stored in the IWF file as external reference entities (X file names). The materials and texture names used by this scene will be stored in the referenced X files, so technically they do not need to be loaded for the IWF. However, we make the calls anyway to allow for scenes that do store such information. They simply load textures and materials stored in the CFileIWF object and add them to the scene texture and material arrays.

```
        // Process the materials and textures first (order is important)
        if (!ProcessMaterials( File )) return false;
        if (!ProcessTextures( File )) return false;
```

As you might expect, the same logic holds true for geometry data. Since the IWF file in this demo does not store any meshes, the function will actually wind up doing nothing, but we implement it anyway to allow for cases where mesh data is exported.

```
        // Process the pure mesh data
        if (!ProcessMeshes( File )) return false;
```

Now we are ready to parse the entities stored in the CFileIWF::m_vpEntityList. This job falls to the CScene::ProcessEntities function. While we have looked at this function in previous demos, its only responsibility was extracting scene lights. This time however, we have two new entity types that we will need to write code for: references and skyboxes. In this demo, our reference entities will contain the filename for the X file(s) we wish to load to create the actual meshes in the scene. The skybox entity will store a list of the six required textures required.

```
        // Copy over the entities we want from the file
        if (!ProcessEntities( File )) return false;
```

Now that we have extracted all of the information that was stored in the IWF file, we can release the memory used by the CFileIWF STL vectors.

```
        // Allow file loader to release any active objects
        File.ClearObjects();
```

If the m_nLightCount member variable is still set to 0 at this point, then the IWF file contained no lighting information. To ensure that we are able to see our objects in this case, we setup four default directional lights and add them to the scene light array. This is not something that you must do but we do it here for convenience.

```
        // If no lights were loaded, lets default some half-way decent ones
        if ( m_nLightCount == 0 )
        {
            // Set up an arbitrary set of directional lights
            ZeroMemory( m_pLightList, 4 * sizeof(D3DLIGHT9));
            m_pLightList[0].Type = D3DLIGHT_DIRECTIONAL;
            m_pLightList[0].Diffuse = D3DXCOLOR( 1.0f, 0.92f, 0.82f, 0.0f );
            m_pLightList[0].Specular = D3DXCOLOR( 1.0f, 0.92f, 0.82f, 0.0f );
            m_pLightList[0].Direction = D3DXVECTOR3( 0.819f, -0.573f, 0.0f );

            m_pLightList[1].Type = D3DLIGHT_DIRECTIONAL;
            m_pLightList[1].Diffuse = D3DXCOLOR( 0.4f, 0.4f, 0.4f, 0.0f );
            m_pLightList[1].Specular = D3DXCOLOR( 0.6f, 0.6f, 0.6f, 0.0f );
            m_pLightList[1].Direction = D3DXVECTOR3( -0.819f, -0.573f, -0.0f );

            m_pLightList[2].Type = D3DLIGHT_DIRECTIONAL;
            m_pLightList[2].Diffuse = D3DXCOLOR( 0.8f, 0.8f, 0.8f, 0.0f );
            m_pLightList[2].Specular = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 0.0f );
            m_pLightList[2].Direction = D3DXVECTOR3( 0.0f, 0.707107f, -0.707107f );

            m_pLightList[3].Type = D3DLIGHT_DIRECTIONAL;
            m_pLightList[3].Diffuse = D3DXCOLOR( 0.6f, 0.6f, 0.6f, 0.0f );
            m_pLightList[3].Specular = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 0.0f );
```

```
            m_pLightList[3].Direction = D3DXVECTOR3( 0.0f, 0.707107f, 0.707107f );

            // We're now using 4 lights
            m_nLightCount = 4;

        } // End if no lights

    } // End Try Block

    // Catch any exceptions
    catch (...)
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

CScene::ProcessMaterials

This function (called by the CScene::LoadSceneFromIWF) loops through each material in the CFileIWF::m_vpMaterialList vector and copies the values into the scene material list. The internal material counter is incremented for each material copied.

```
bool CScene::ProcessMaterials( const CFileIWF& File )
{
    ULONG i;

    // Loop through and build our materials
    for ( i = 0; i < File.m_vpMaterialList.size(); i++ )
    {
        // Retrieve pointer to file material
        iwfMaterial * pFileMaterial = File.m_vpMaterialList[i];

        // Retrieve pointer to our local material
        D3DMATERIAL9 * pMaterial = &m_pMaterialList[i];

        // Copy over the data we need from the file material
        pMaterial->Diffuse  = (D3DCOLORVALUE&)pFileMaterial->Diffuse;
        pMaterial->Ambient  = (D3DCOLORVALUE&)pFileMaterial->Ambient;
        pMaterial->Emissive = (D3DCOLORVALUE&)pFileMaterial->Emissive;
        pMaterial->Specular = (D3DCOLORVALUE&)pFileMaterial->Specular;
        pMaterial->Power    = pFileMaterial->Power;

        // Increase internal vars
        m_nMaterialCount++;
        if ( m_nMaterialCount >= MAX_MATERIALS ) break;

    } // Next Material

    // Success!
    return true;
}
```

CScene::ProcessTextures

This function (called by CScene::LoadScene from IWF) loops through the CFileIWF::m_vpTextureList and uses the stored filenames to load the texture resources. This function does not need to worry about duplicates because IWF files only store unique texture filenames. As each texture is loaded, its IDirect3DTexture9 interface pointer and filename is stored in the scene objects m_pTextureList array.

The first thing we do is zero out the scene TEXTURE_ITEM array.

```
bool CScene::ProcessTextures( const CFileIWF& File )
{
    ULONG          i;
    TCHAR          Buffer[MAX_PATH];
    TEXTURE_ITEM * pNewTexture = NULL;

    ZeroMemory( m_pTextureList, m_nTextureCount * sizeof(TEXTURE_ITEM*));
```

Now we will loop through each TEXTURE_REF in m_vpTextureList. The TEXTURE_REF structure is defined in libIWF.h and contains the texture filename and possibly even the actual texture pixel data. We are interested only in the filename (and its string length) for now.

```
for ( i = 0; i < File.m_vpTextureList.size(); i++ )
    {
        // Retrieve pointer to file texture
        TEXTURE_REF * pFileTexture = File.m_vpTextureList[i];

        // Skip if this is an internal texture (not supported by this demo)
        if ( pFileTexture->TextureSource != TEXTURE_EXTERNAL ) continue;
```

As we have found a potentially valid texture, we will create a new TEXTURE_ITEM to store the filename and texture we are about to create. We add the item to the CScene::m_pTextureList array.

```
        // No texture found, lets create our texture data and store it
        pNewTexture = new TEXTURE_ITEM;
        ZeroMemory( pNewTexture, sizeof(TEXTURE_ITEM) );
```

The TEXTURE_REF structure stores only the filename and not the full path. This allows us to store the texture in any folder desired. We extracted the path information during the LoadSceneFromIWF function, so we can add the path string to the texture file name to get the full path for the texture for loading. We use D3DXCreateTextureFromFileEx to load the texture.

```
        // Now build the full path
        _tcscpy( Buffer, m_strDataPath );
        _tcscat( Buffer, pFileTexture->Name );

        // Create the texture (use 3 mip levels max) (Ignore error value)
        D3DXCreateTextureFromFileEx( m_pD3DDevice, Buffer, D3DX_DEFAULT, D3DX_DEFAULT, 3, 0,
                                     m_fmtTexture,D3DPOOL_MANAGED,D3DX_DEFAULT,D3DX_DEFAULT,
                                     0, NULL, NULL, &pNewTexture->Texture );
```

We copy the texture filename into the TEXURE_ITEM, add the structure to our global array, and then increment our counter.

```
        // Duplicate the filename for future lookups
        pNewTexture->FileName = _tcsdup( pFileTexture->Name );

        // Store this item
        m_pTextureList[ m_nTextureCount++ ] = pNewTexture;
        if ( m_nTextureCount >= MAX_TEXTURES ) break;

    } // Next Texture

    // Success!
    return true;
}
```

CScene::ProcessMeshes

ProcessMeshes extracts geometry from CFileIWF::m_vpMeshList and creates CTriMesh objects for each mesh. The m_vpMeshList vector contains one iwfMesh structure for each mesh extracted from the file (see libIWF.h).

```
bool CScene::ProcessMeshes( CFileIWF & pFile )
{
    HRESULT     hRet;
    CTriMesh * pNewMesh = NULL;
    long        i, j, MaterialIndex, TextureIndex;
    ULONG       AttribID = 0;

    // Loop through each mesh in the file
    for ( i = 0; i < pFile.m_vpMeshList.size(); i++ )
    {
        iwfMesh * pMesh = pFile.m_vpMeshList[i];

        // Allocate a new CTriMesh
        pNewMesh = new CTriMesh;
        if ( !pNewMesh ) return false;
```

We are going to populate mesh data buffers manually, so we need to tell the mesh about our vertex components and our index format so that it can properly allocate its vertex/index buffers. In this project, VERTEX_FVF is defined in CObject.h as:

```
#define VERTEX_FVF      D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1
```

```
        // Set the mesh's data format
        pNewMesh->SetDataFormat( VERTEX_FVF, sizeof(USHORT) );
```

Now that we have created the CTriMesh, we can loop through each surface in the iwfMesh and add the geometry data. Note that we skip surfaces marked with the SURFACE_INVISIBLE flag.

```
        for ( j = 0; j < pMesh->SurfaceCount; j++ )
```

```
        {
            iwfSurface * pSurface = pMesh->Surfaces[j];

            // Skip if this surface is flagged as invisible
            if ( pSurface->Style & SURFACE_INVISIBLE ) continue;
```

First we extract the texture and material indices used by the face setting the indices to –1 if some error has occurred (e.g., the texture index has a larger value than the total number of textures stored in the IWF file). This would mean the IWF file has been created incorrectly. We do not halt execution but instead assign an idex of –1 indicating that the scene's default (white) material will need to be assigned to these faces.

```
           // Determine the indices we are using.
           MaterialIndex = -1;
           TextureIndex  = -1;
          if ( (pSurface->Components & SCOMPONENT_MATERIALS) && pSurface->ChannelCount > 0 )
             MaterialIndex = pSurface->MaterialIndices[0];

          if ( (pSurface->Components & SCOMPONENT_TEXTURES ) && pSurface->ChannelCount > 0 )
             TextureIndex  = pSurface->TextureIndices[0];

          if ( MaterialIndex >= m_nMaterialCount ) MaterialIndex = -1;
          if ( MaterialIndex >= m_nTextureCount  ) TextureIndex  = -1;
```

With the material and texture indices stored in temporary local variables (MaterialIndex and TextureIndex), we use them to get a pointer to the material in the scene material array and the texture filename for the texture stored in the scene texture array (shown below).

```
        LPCTSTR        strTextureFile = NULL;
        D3DMATERIAL9 * pMaterial      = NULL;

        // Retrieve information to pass to support functions
        if ( MaterialIndex >= 0 ) pMaterial = &m_pMaterialList[MaterialIndex];
        if ( TextureIndex  >= 0 ) strTextureFile = m_pTextureList[TextureIndex]->FileName;
```

Bear in mind when looking at the above code that when this function has been called, the ProcessMaterials and ProcessTextures function have already been executed. Therefore, all textures and materials stored inside the IWF file are already in the scene texture and material lists at this point. This means the texture and material index stored in each IWFSurface indexes correctly into the scene's texture and material arrays. This is because they will have been added to the scene in the same order as the materials and textures listed inside the IWF file. The texture and material indices stored in each IWFSurface are therefore still valid when used to access the scene textures and materials.

Now that we have a pointer to the material and the texture filename used by this face, we pass this information into the CScene::CollectAttributeID function. The function will look for a matching material/texture combination in its ATTRIBUTE_ITEM array and if one is not found, create a new entry, loading and creating any textures and materials as necessary. This is the same function that is used as a callback function for non-managed mode meshes when loading X file data.

The function returns the index of the ATTRIBUTE_ITEM which describes the global attribute ID that will be assigned to this face when it is added to the CTriMesh. Notice that we also pass in the 'this' pointer because CollectAttributeID is a static function.

```
        // Collect an attribute ID
        AttribID = CollectAttributeID( this, strTextureFile, pMaterial );
```

Now that we have the attribute index for this material/texture combination, we can add the face indices to the CTriMesh index buffer. For each triangle added (because this may be an N-gon), we also copy the attribute ID into the CTriMesh attribute buffer so that all triangles that share the same material and texture will belong to the same subset. This is all handled by the ProcessIndices function. We pass in a pointer to our new CTriMesh, a pointer to the iwfSurface that contains the index data for the face we wish to add, and the face attribute ID that we have just generated.

```
        // Process the indices
        if (!ProcessIndices( pNewMesh, pSurface, AttribID ) ) break;
```

If the surface has the SURFACE_TWO_SIDED flag set, then the level designer wants this surface to be visible from both its front and back sides. As our application uses back face culling, we add the face to the mesh again, this time passing in TRUE as the final parameter. This call reverses the winding order of the face before it adds it to the index buffer the second time. We have essentially created two polygons that share the same position in 3D space but face in opposing directions.

```
        if ( pSurface->Style & SURFACE_TWO_SIDED )
        {
            // Two sided surfaces have back faces added manually
            if (!ProcessIndices( pNewMesh, pSurface, AttribID, true ) ) break;
        }
```

Now we call ProcessVertices to add the vertex data to the new mesh.

```
           // Process vertices
           if (!ProcessVertices( pNewMesh, pSurface ) ) break;

    } // Next Surface
```

After the CScene::ProcessVertices and CScene::ProcesIndices have been called, the CTriMesh will have had all the vertex, index and attribute data added to its temporary storage bins.

If one of the processing functions failed, then the loop variable 'j' will be smaller than the surface count of the iwfMesh. If this is the case, then we release the new CTriMesh and return failure as we have data corruption.

```
        // If we didn't reach the end, then we failed
        if ( j < pMesh->SurfaceCount ) { delete pNewMesh; return false; }
```

We next instruct the CTriMesh object to build its underlying ID3DXMesh with a call to the CTriMesh::BuildMesh function. We pass in the D3DXMESH_MANAGED creation flag indicating our desire for allocating index and vertex buffers in the managed resource pool.

```
        // We're done, attempt to build this mesh
        hRet = pNewMesh->BuildMesh( D3DXMESH_MANAGED, m_pD3DDevice );
        if ( FAILED(hRet) ) { delete pNewMesh; return false; }
```

Next we clean and optimize the CTriMesh. We start with the CTriMesh::Weld function, passing in a 0.0 floating point tolerance so that only exact duplicated vertices are merged. Then we call the CTriMesh::OptimizeInPlace function to perform compaction, attribute sorting, and vertex cache optimizations on the underlying D3DXMesh data.

```
        // Optimize the mesh of possible
        pNewMesh->WeldVertices( 0 );
        pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );
```

We now call the CScene::AddMesh() function, which is a simple utility function to resize the scene CTriMesh array (m_pMesh). This call returns the index of the newly added element, which we use to store the CTriMesh pointer to the mesh we have just created.

```
        // Store this new mesh
        if ( AddMesh( ) < 0 ) { delete pNewMesh; return false; }
        m_pMesh[ m_nMeshCount - 1 ] = pNewMesh;
```

Although we have added the mesh to the scene, we still require a higher level object to allow for mesh instancing, so we allocate a new CObject structure, passing our new mesh into the constructor. Note that while the CObject maintains a world matrix, meshes stored in IWF files exported from GILES™ are all defined in world space. So the CObject world matrix should just be set to identity, and this is done implicitly by the constructor.

```
        // Now build an object for this mesh (standard identity)
        CObject * pNewObject = new CObject( pNewMesh );
        if ( !pNewObject ) return false;
```

Finally, we add this new CObject to the scene level CObject array (m_pObject). The AddObject() member function resizes the array and returns the index where we will copy the pointer .

```
        // Store this object
        if ( AddObject() < 0 ) { delete pNewObject; return false; }
        m_pObject[ m_nObjectCount - 1 ] = pNewObject;

    } // Next Mesh

    // Success!!
    return true;
}
```

CScene::ProcessVertices

ProcessVertices is called by ProcessMeshes (discussed above) for each surface in the mesh. Its job is to add the vertices stored in the iwfSurface to the newly created CTriMesh object. Note that they are not copied straight into the vertex buffer because the ID3DXMesh will not have been created yet. Instead they are copied into the CTriMesh::pVertices array. Only after all vertices have been added and the underlying ID3DXMesh has been created will we fill the vertex buffer with this data.

```
bool CScene::ProcessVertices( CTriMesh * pMesh, iwfSurface * pFilePoly )
{
    ULONG i, VertexStart = pMesh->GetNumVertices();
    CVertex * pVertices = NULL;

    // Allocate enough vertices
    if ( pMesh->AddVertex( pFilePoly->VertexCount ) < 0 ) return false;
    pVertices = (CVertex*)pMesh->GetVertices();
```

We start by retrieving the number of vertices currently stored in the mesh. This tells us where we want to start appending our new vertex data. Next we call the CTriMesh::AddVertex function passing in the number of vertices in the iwfSurface. This function resizes the mesh's temporary vertex array if necessary to make room for the new data. We then get a pointer to the mesh vertex array so that we can start copying the vertex data.

```
    // Loop through each vertex and copy required data.
    for ( i = 0; i < pFilePoly->VertexCount; i++ )
    {
        // Copy over vertex data
        pVertices[i + VertexStart].x      = pFilePoly->Vertices[i].x;
        pVertices[i + VertexStart].y      = pFilePoly->Vertices[i].y;
        pVertices[i + VertexStart].z      = pFilePoly->Vertices[i].z;
        pVertices[i + VertexStart].Normal = (D3DXVECTOR3&)pFilePoly->Vertices[i].Normal;

        // If we have any texture coordinates, set them
        if ( pFilePoly->TexChannelCount > 0 && pFilePoly->TexCoordSize[0] == 2 )
        {
            pVertices[i + VertexStart].tu = pFilePoly->Vertices[i].TexCoords[0][0];
            pVertices[i + VertexStart].tv = pFilePoly->Vertices[i].TexCoords[0][1];
        } // End if has tex coordinates
    } // Next Vertex

    // Success!
    return true;
}
```

CScene::ProcessIndices

ProcessIndices remains relatively unchanged from our prior projects so we will only show the code that deals with face indices in indexed triangle list format. This means we simply copy the indices from the passed iwfSurface into the mesh. Please refer to Chapter Five for code that converts the other formats.

```
bool CScene::ProcessIndices( CTriMesh * pMesh, iwfSurface * pFilePoly,
                             ULONG AttribID, bool BackFace )
{
    ULONG    i, Counter, VertexStart, FaceStart, IndexStart;
    USHORT * pIndices = NULL;

    // Validate parameters
    if (!pMesh || !pFilePoly) return false;
```

Before we add indices to the new CTriMesh object, we need to retrieve the mesh vertex count so that we can correctly offset the zero-based index values. Keep in mind that this function is called before the vertices are processed, so if we did not do this, regardless of how many vertices we added, each triangle would refernce the first three vertices. We also also fetch the number of triangles currently stored in the CTriMesh so that we can correctly calculate the offset in the mesh index array for appending the new indices.

```
    // Store current Mesh vertex and face count
    VertexStart = pMesh->GetNumVertices();
    FaceStart   = pMesh->GetNumFaces();
    IndexStart  = FaceStart * 3;
```

If the iwfSurface has a non-zero index count then the face includes indices. If not, then the function will need to generate indices for the appropriate polygon type (see Chapter Five). All we have to do for the current case we are looking at is copy the index data.

```
    // Generate indices
    if ( pFilePoly->IndexCount > 0 )
    {
        ULONG IndexType = pFilePoly->IndexFlags & INDICES_MASK_TYPE;

        // Interpret indices (we want them in tri-list format)
        switch ( IndexType )
        {
             case INDICES_TRILIST:

                // We can do a straight copy (converting from 32bit to 16bit)
                if ( pMesh->AddFace( pFilePoly->IndexCount / 3, NULL, AttribID ) < 0 )
                       return false;

                pIndices = (USHORT*)pMesh->GetFaces( );

                // Copy over the face data
                for ( i = 0; i < pFilePoly->IndexCount; ++i )
                       pIndices[i + IndexStart] = pFilePoly->Indices[i] + VertexStart;

             break;
```

After the indices have been added, we test the BackFace Boolean passed into the function. If TRUE, then we need to reverse the order of the indices just added and re-copy them. This approach is used when we encounter a two sided face in the IWF file.

```
  // We now have support for adding the same polygon again, but in
  // reverse order to add a renderable back face if disabling culling
  // is not a viable option.
  if ( BackFace == true )
  {
        // If we specified back faces, invert all the indices recently added
               pIndices  = (USHORT*)pMesh->GetFaces( );
        for ( i = IndexStart; i < pMesh->GetNumFaces() * 3; i+=3 )
        {
               USHORT iTemp  = pIndices[i];
               pIndices[i]   = pIndices[i+2];
               pIndices[i+2] = iTemp;

        } // Next Tri

  } // End if back face

    // Success!
    return true;
}
```

CScene::CollectAttributeID

This function is called from two possible places. CTriMesh::LoadMeshFromX calls it to process a material/texture combination extracted from an X file. CScene::ProcessMeshes calls it to process a material/texture combination used by an iwfSurface. The function is responsible for making sure that textures and materials are not duplicated and it returns the global attribute ID for subsets in non-managed meshes.

The first parameter is a void pointer to a context. Remember that this function was declared as a static member function so that it can double as a callback function for non-managed mode meshes when needed. We use the context parameter in the ProcessMeshes function to pass the ‘this’ pointer so that there can be access to non-static member variables. The next three parameters are what we will match against when searching for the appropriate combination of attributes.

```
ULONG CScene::CollectAttributeID( LPVOID pContext, LPCTSTR strTextureFile,
                                  const D3DMATERIAL9 * pMaterial,
                                  const D3DXEFFECTINSTANCE * pEffectInstance )
{
    TEXTURE_ITEM * pNewTexture = NULL;
    ULONG          i;
    long           TextureMatch = -1, MaterialMatch = -1;
    bool           TexMatched = false, MatMatched = false;

    // Validate parameters
    if ( !pContext ) return 0;

    // Retrieve the scene object
    CScene *pScene = (CScene*)pContext;
```

We cast the context pointer to a CScene pointer so that we have access to the texture and materials arrays for the correct scene object instance. We can now loop through the scene ATTRIBUTE_ITEM array to see if an element with the passed texture/material combination already exists. The process should be relatively easy to follow, so we will not explain it in great detail. We start by checking texture filenames and then the material pointers.

```
    // Loop through the attribute combination table to see if one already exists
    for ( i = 0; i < pScene->m_nAttribCount; ++i )
    {
        ATTRIBUTE_ITEM * pAtrItem = &pScene->m_pAttribCombo[i];
        long TextureIndex  = pAtrItem->TextureIndex;
        long MaterialIndex = pAtrItem->MaterialIndex;

        TEXTURE_ITEM   * pTexItem = NULL;
        D3DMATERIAL9   * pMatItem = NULL;

        // Retrieve pointers
        if ( TextureIndex  >= 0 ) pTexItem = pScene->m_pTextureList[ TextureIndex ];
        if ( MaterialIndex >= 0 ) pMatItem = &pScene->m_pMaterialList[ MaterialIndex ];

        // Neither are matched so far
        TexMatched = false;
        MatMatched = false;

        // If both sets are NULL, this is a match, otherwise perform the real match
        if ( pTexItem == NULL && strTextureFile == NULL )
            TexMatched = true;

        else if ( pTexItem != NULL && strTextureFile != NULL )
        if ( _tcsicmp( pTexItem->FileName, strTextureFile ) == 0 ) TexMatched = true;

        if ( pMatItem == NULL && pMaterial == NULL )
            MatMatched = true;
        else if ( pMatItem != NULL && pMaterial != NULL )

        if ( memcmp( pMaterial, pMatItem, sizeof(D3DMATERIAL9) ) == 0 ) MatMatched = true;

        // Store the matched indices in case we can use the later on
        if ( TexMatched ) TextureMatch = TextureIndex;
        if ( MatMatched ) MaterialMatch = MaterialIndex;
```

If we have both a texture match and a material match, then we return the index of this attribute item to the caller. The CTriMesh::LoadMeshFromX function uses this index to remap its attribute buffer from mesh local attribute IDs to scene attribute IDs.

```
        // If they both matched up at the same time, we have a winner
        if ( TexMatched == true && MatMatched == true ) return i;

    } // Next Attribute
```

If we cannot find a match on both parameters (texture and material), then we will need to add a new ATTRIBUTE_ITEM structure to the scene array reflecting these attribute properties. We want to look

for individual matches on texture or material first to avoid duplicating data, so we will traverse those arrays again and record the index of the matches if we find them.

```
    ATTRIBUTE_ITEM AttribItem;

    if ( MaterialMatch < 0 && pMaterial != NULL )
    {
        for ( i = 0; i < pScene->m_nMaterialCount; ++i )
        {
            // Is there a match ?
            if ( memcmp( pMaterial, &pScene->m_pMaterialList[i], sizeof(D3DMATERIAL9)) == 0 )
                 MaterialMatch = i;

            // If we found a match, bail
            if ( MaterialMatch >= 0 ) break;

        } // Next Attribute Combination

    } // End if no material match

    if ( TextureMatch < 0 && strTextureFile != NULL )
    {
        for ( i = 0; i < pScene->m_nTextureCount; ++i )
        {
            if (!pScene->m_pTextureList[i] || !pScene->m_pTextureList[i]->FileName) continue;

            // Is there a match ?
            if ( _tcsicmp( strTextureFile, pScene->m_pTextureList[i]->FileName ) == 0 )
                 TextureMatch = i;

            // If we found a match, bail
            if ( TextureMatch >= 0 ) break;

        } // Next texture

    } // End if no Texture match
```

If matches are found (for either type), we simply copy the index value from the local variable into the new ATTRIBUTE_ITEM. When a match is not found, we must load the data into our global lists and record those indices instead. Materials will simply be copied, textures will require loading.

```
    // Now build the material index, or add if necessary
    if ( MaterialMatch < 0 && pMaterial != NULL )
    {
        AttribItem.MaterialIndex = pScene->m_nMaterialCount;
        pScene->m_pMaterialList[ pScene->m_nMaterialCount++ ] = *pMaterial;
    } // End if no material match
    else
    {
        AttribItem.MaterialIndex = MaterialMatch;
    } // End if material match
```

If a matching texture was not found in the scene texture array then we need to create a new texture with a call to the CollectTexture function. This function (also used as the texture callback function for managed mode meshes in the CTriMesh::LoadMeshFromX function) will load the texture if it does not yet exist

and will add it to the end of the scene texture array. If a texture was found, we copy this index into the attribute item structure as shown below.

```
    // Now build the texture index, or add if necessary
    if ( TextureMatch < 0 && strTextureFile != NULL )
    {
        // We know it doesn't exist, but we can still use
        // collect texture to do the work for us.
        AttribItem.TextureIndex = pScene->m_nTextureCount;
        CollectTexture( pScene, strTextureFile );

    } // End if no texture match
    else
    {
        AttribItem.TextureIndex = TextureMatch;

    } // End if Texture match
```

Finally, we add the new item to the scene m_pAttribCombo array and return the index of this new attribute.

```
    // Store this new attribute combination item
    pScene->m_pAttribCombo[ pScene->m_nAttribCount++ ] = AttribItem;

    // Return the new attribute index
    return pScene->m_nAttribCount - 1;
}
```

CScene::CollectTexture

This function searches the scene texture array to determine whether a texture with the passed texture filename already exists. If so, a pointer to this texture item is returned. Otherwise, a new texture is created, inserted into the array, and its pointer is returned.

CollectTexture is called from a number of places in our code. If we wish to load mesh data from an X file into a managed mode CTriMesh, we can register this function as the texture callback so that texture filenames returned from D3DXLoadMeshFromX can be mapped to textures in our scene texture array. The CollectAttributeID function, which can serve as a callback for non-managed meshes, calls this function to load its textures as well. We have just seen how ProcessMeshes calls the CollectAttributeID function (which in turn calls CollectTexture) to add the textures in an IWF file to the scene texture list.

```
LPDIRECT3DTEXTURE9 CScene::CollectTexture( LPVOID pContext, LPCTSTR FileName )
{
    HRESULT        hRet;
    TEXTURE_ITEM * pNewTexture = NULL;

    // Validate parameters
    if ( !pContext || !FileName ) return NULL;

    // Retrieve the scene object
```

```
    CScene *pScene = (CScene*)pContext;
```

Once we have a pointer to the scene instance, we can loop through its texture array and compare each texture name with the texture name passed into the function. If a match is found then the texture already exists and we immediately return a pointer to the texture.

```
    // Loop through and see if this texture already exists.
    for ( ULONG i = 0; i < pScene->m_nTextureCount; ++i )
    {
        if ( _tcsicmp( pScene->m_pTextureList[i]->FileName, FileName ) == 0 )
            return pScene->m_pTextureList[i]->Texture;

    } // Next Texture
```

If we exit the loop, then we know that a texture with a matching filename does not yet exist. Therefore, we need to create and initialize a new TEXTURE_ITEM structure that will be added to the texture array. Note that the texture filename passed into the function will not contain any path information. So in order to load the image file into a texture, we build the complete filename string with the full path. We will store the texture filename without a path in the TEXTURE_ITEM structure for future searches.

```
    // No texture found, so lets create and store it.
    pNewTexture = new TEXTURE_ITEM;
    if (!pNewTexture) return NULL;
    ZeroMemory( pNewTexture, sizeof(TEXTURE_ITEM) );

    // Build filename string
    TCHAR Buffer[MAX_PATH];
    _tcscpy( Buffer, pScene->m_strDataPath );
    _tcscat( Buffer, FileName);

    // Create the texture (use 3 mip levels max)
    hRet = D3DXCreateTextureFromFileEx(pScene->m_pD3DDevice, Buffer, D3DX_DEFAULT,
                                       D3DX_DEFAULT, 3, 0, pScene->m_fmtTexture,
                                       D3DPOOL_MANAGED, D3DX_DEFAULT, D3DX_DEFAULT,
                                       0, NULL, NULL, &pNewTexture->Texture );

    if (FAILED(hRet)) { delete pNewTexture; return NULL; }

    // Duplicate the filename for future lookups
    pNewTexture->FileName = _tcsdup( FileName );
```

Finally we add the new TEXTURE_ITEM to the scene array, increment the texture count, and return the new texture pointer.

```
    // Store this item
    pScene->m_pTextureList[ pScene->m_nTextureCount++ ] = pNewTexture;

    // Return the texture pointer
    return pNewTexture->Texture;
}
```

CScene::ProcessEntities

Our current project will support three different entity types: lights, skyboxes, and references. This function is responsible for navigating the CFileIWF::m_vpEntityList vector and extracting these types as they are encountered.

The first part of the function tests for light entities. The light entity is one of the IWF standard entity types and it is defined in the IWF SDK as an entity with an ID of 16:

```
#define ENTITY_LIGHT                        0x0010
```

```
bool CScene::ProcessEntities( const CFileIWF& File )
{
    ULONG      i, j;
    D3DLIGHT9  Light;
    USHORT     StringLength;
    bool       SkyBoxBuilt = false;

    // Loop through and build our lights & references
    for ( i = 0; i < File.m_vpEntityList.size(); i++ )
    {
        // Retrieve pointer to file entity
        iwfEntity * pFileEntity = File.m_vpEntityList[i];

        // Skip if there is no data
        if ( pFileEntity->DataSize == 0 ) continue;
```

We will extract light information from the entity data area directly into a D3DLIGHT9 structure and add the new light to our scene light array. In this project we will only process the light if we have not yet added the maximum number of lights to the array, otherwise it will be ignored.

```
        // Decide what to do here (skip all lights if we've already reached our limit)
        if ( (m_nLightCount < m_nLightLimit && m_nLightCount < MAX_LIGHTS)
              && pFileEntity->EntityTypeMatches( ENTITY_LIGHT ) )
        {
            LIGHTENTITY * pFileLight = (LIGHTENTITY*)pFileEntity->DataArea;
```

GILES™ can export ambient lights, but they are not particularly relevant in the DirectX lighting pipeline, so if the current light is an ambient light we will skip it

```
            // Skip if this is not a valid light type (Not relevant to the API)
            if ( pFileLight->LightType == LIGHTTYPE_AMBIENT ) continue;
```

Extracting the lighting information is very simple due to the fact that the LIGHTENTITY structure used by CFileIWF is almost identical to the D3DLIGHT9 structure.

```
            // Extract the light values we need
            Light.Type      = (D3DLIGHTTYPE)(pFileLight->LightType + 1);
            Light.Diffuse   = D3DXCOLOR(pFileLight->DiffuseRed, pFileLight->DiffuseGreen,
                                        pFileLight->DiffuseBlue, pFileLight->DiffuseAlpha);
```

```
            Light.Ambient   = D3DXCOLOR (pFileLight->AmbientRed, pFileLight->AmbientGreen,
                                         pFileLight->AmbientBlue, pFileLight->AmbientAlpha);

            Light.Specular  = D3DXCOLOR (pFileLight->SpecularRed, pFileLight->SpecularGreen,
                                         pFileLight->SpecularBlue,pFileLight->SpecularAlpha);
```

Every entity stores a world matrix, so we extract the bottom row to get the light position in world space.

```
            Light.Position  = D3DXVECTOR3(pFileEntity->ObjectMatrix._41,
                                          pFileEntity->ObjectMatrix._42,
                                          pFileEntity->ObjectMatrix._43 );
```

We also extract the look vector which describes the direction the light is pointing in world space.

```
            Light.Direction  = D3DXVECTOR3(pFileEntity->ObjectMatrix._31,
                                           pFileEntity->ObjectMatrix._32,
                                           pFileEntity->ObjectMatrix._33 );
```

The remaining light data is extracted and we add the light to our light array, incrementing the counter.

```
            Light.Range            = pFileLight->Range;
            Light.Attenuation0     = pFileLight->Attenuation0;
            Light.Attenuation1     = pFileLight->Attenuation1;
            Light.Attenuation2     = pFileLight->Attenuation2;
            Light.Falloff          = pFileLight->FallOff;
            Light.Theta            = pFileLight->Theta;
            Light.Phi              = pFileLight->Phi;

            // Add this to our array
            m_pLightList[ m_nLightCount++ ] = Light;

        } // End if light
```

Testing for skyboxes and reference entities requires a slightly different approach because these types are not defined by the IWF standard -- they are custom entity types defined by GILES™. The IWF specification provides applications the ability to specify their own entity types with their own entity IDs and chunk IDs provided they do not conflict with IDs reserved by the IWF standard.

To address the possibility of third party entities sharing the same IDs with each other, the IWF specification records an author ID in addition to the entity chunk IDs. For example, the skybox and reference entities both have the author ID for GILES™ embedded in those chunks. GILES™ uses a 5 BYTE author ID embedded in its custom chunks. Each byte is the ASCII code for the letters of its name. We define this array in CScene.cpp and we can use it to test if an entity is a custom GILES™ entity.

```
const UCHAR AuthorID[5] = { 'G', 'I', 'L', 'E', 'S' };
```

The iwfEntity class contains a helper function called EntityAuthorMatches. We pass in an array length and an array of bytes which will be compared against the entity's author ID, returning true if they match.

If the entity is not a light entity or a GILES™ custom entity (skybox/reference) we will ignore it.

```
        else if ( pFileEntity->EntityAuthorMatches( 5, AuthorID ) )
        {
```

At this point we do not know if this will be a skybox entity of a reference entity, so we will create two local variables that can be used to hold the information for both.

```
            SkyBoxEntity SkyBox;
            ZeroMemory( &SkyBox, sizeof(SkyBoxEntity) );

            ReferenceEntity Reference;
            ZeroMemory( &Reference, sizeof(ReferenceEntity ));
```

The SkyBoxEntity and ReferenceEntity structures are both defined in CScene.h.

```
typedef struct _ReferenceEntity
{
    ULONG       ReferenceType;          // What type of reference is this
    ULONG       Flags;                  // Reserved flags
    ULONG       Reserved1;              // Reserved values
    ULONG       Reserved2;              // Reserved values
    ULONG       Reserved3;              // Reserved values
    char        ReferenceName[1024];    // External file name

} ReferenceEntity;
```

The entity ID for the GILES™ reference entity is 0x203 (CScene.h).

```
#define CUSTOM_ENTITY_REFERENCE 0x203
```

Most of the members of the reference entity are reserved for later use. We are interested only in the reference type (0 for internal reference, 1 for external reference) and the ReferenceName. In this project we will work with external references only. The name of the referenced X file will be stored in the ReferenceName member.

The skybox entity stores a reserved DWORD followed by six texture filenames. When our application encounters one of these entities, we create the scene skybox mesh (m_SkyBoxMesh) as an inward facing cube, and then load the six textures using these filenames and map them to the cube faces.

```
typedef struct _SkyBoxEntity
{
    ULONG       Flags;                  // Reserved flags
    char        Textures[6][256];       // 6 Sets of external texture names
} SkyBoxEntity;
```

The entity ID for the GILES™ skybox entity is 0x202 (CScene.h).

```
#define CUSTOM_ENTITY_SKYBOX    0x202
```

To determine which entity type we have found, we retrieve a pointer to the entity data area and check its ID.

```
            // Retrieve data area
            UCHAR * pEntityData = pFileEntity->DataArea;
```

If it is a GILES™ reference entity then we extract the data into the temporary ReferenceEntity structure and pass it to CScene::ProcessReference for additional processing. Notice that the ProcessReference function accepts the ReferenceEntity and the entity world matrix. This matrix stores the world space position and orientation of the object that will ultimately be added to the scene for this reference. Every IWF entity contains a matrix describing the position and orientation of the entity in the scene. The following code shows how to extract all the entity information and pass it along to the ProcessReference function.

```
            switch ( pFileEntity->EntityTypeID )
            {
                case CUSTOM_ENTITY_REFERENCE:

                    // Copy over the the reference data
                    memcpy( &Reference.ReferenceType, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Reference.Flags, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &StringLength, pEntityData, sizeof(USHORT) );
                    pEntityData += sizeof(USHORT);

                    if ( StringLength > 0 )
                       memcpy( Reference.ReferenceName, pEntityData, StringLength );
                    pEntityData += StringLength;

                    memcpy( &Reference.Reserved1, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Reference.Reserved2, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Reference.Reserved3, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    // Process this reference (returns false only on fatal error)
                    if(!ProcessReference(Reference,(D3DXMATRIX&)pFileEntity->ObjectMatrix))
                        return false;

                    break;
```

For skybox entities we use a similar approach. We copy the data into the temporary SkyBoxEntity structure and pass it to ProcessSkyBox for final processing. Note that we only process the sky box entity if a sky box does not currently exist. This was done for simplicity only, so feel free to change this behavior to maintain an array of skyboxes that can be modified as you wish. Usually, a given scene will have one sky box defined, but this is not always necessarily the case.

```
                case CUSTOM_ENTITY_SKYBOX:
```

```
                    // We only want one skybox per file please! :)
                    if ( SkyBoxBuilt == true ) break;
                    SkyBoxBuilt = true;

                    // Copy over the skybox data
                    memcpy( &SkyBox.Flags, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    // Read each of the 6 face textures
                    for ( j = 0; j < 6; ++j )
                    {
                        memcpy( &StringLength, pEntityData, sizeof(USHORT) );
                        pEntityData += sizeof(USHORT);

                        if ( StringLength > 0 )
                          memcpy( SkyBox.Textures[j], pEntityData, StringLength );
                        pEntityData += StringLength;

                    } // Next Face Texture

                    // Process this skybox (returns false only on a fatal error)
                    if ( !ProcessSkyBox( SkyBox ) ) return false;

                    break;

            } // End Entity Type Switch

        } // End if custom entities

    } // Next Entity

    // Success!
    return true;
}
```

CScene::ProcessReference

As mentioned previously, the space scene for this project stores all scene objects except for the skybox as external reference entities. These entities contain a world matrix and the filename for an X file which the object will use as its mesh.

The first thing this function does is check the scene mesh array to see if the mesh has already been loaded. This is done because there may be many reference entities which reference the same X file and we would prefer to avoid duplication, to keep memory footprint as small as possible. If the mesh already exists, then we will simply create a new CObject, instance the CTriMesh pointer, and then use the reference matrix for the object's world matrix. If the mesh does not currently exist, then we will need to create a new CTriMesh object and load the mesh data from the X file using the CTriMesh::LoadMeshFromX function, passing in the file name stored in the reference. It is in this function that we will see, for the first time, the registration of callback functions with the CTriMesh object. As discussed, these functions will automatically add the texture and materials in the X file to the scene level texture and material arrays.

The first thing the code does is test the ReferenceType member of the entity. If it is not set to 1 then this is an internal reference and we will ignore this entity and return. If it is set to 0, then we have an external reference, so we will build the full path and filename for the referenced X file (the reference entity will not contain any path information, only the name of the file).

```
bool CScene::ProcessReference(const ReferenceEntity& Reference, const D3DXMATRIX & mtxWorld)
{
    HRESULT     hRet;
    CTriMesh * pReferenceMesh = NULL;

    if (Reference.ReferenceType != 1) return true;

    // Build filename string
    TCHAR Buffer[MAX_PATH];
    _tcscpy( Buffer, m_strDataPath );
    _tcscat( Buffer, Reference.ReferenceName );
```

We now search the scene mesh array for a mesh with the same name as the file name. If a match is found, we will store a pointer to this matching mesh for now.

```
    // Search to see if this X file has already been loaded
    for ( ULONG i = 0; i < m_nMeshCount; ++i )
    {
        if (!m_pMesh[i]) continue;
        if ( _tcsicmp( Buffer, m_pMesh[i]->GetMeshName() ) == 0 ) break;
        } // Next Mesh

    // If we didn't reach the end, this mesh already exists
    if ( i != m_nMeshCount )
    {
        // Store reference mesh.
        pReferenceMesh = m_pMesh[i];

    } // End if mesh already exists
```

If the mesh is not already loaded then we will need to create a new CTriMesh object and call its LoadMeshFromX function to load the X file data into the mesh. Before we do that though, we register the CScene::CollectAttributeID function, which automatically places the mesh into non-managed mode. When the mesh loads the X file data, each texture and material loaded from the X file will be passed to the CollectAttributeID function. We saw earlier how this function adds the texture and material to the scene texture and material arrays and returns the index for this texture/material combo back to the mesh. The mesh will then use this index to remap its attribute buffer.

```
    else
    {
        // Allocate a new mesh for this reference
        CTriMesh * pNewMesh = new CTriMesh;

        // Load in the externally referenced X File
        pNewMesh->RegisterCallback(CTriMesh::CALLBACK_ATTRIBUTEID, CollectAttributeID, this);
        pNewMesh->LoadMeshFromX( Buffer, D3DXMESH_MANAGED, m_pD3DDevice );
```

Once the mesh is loaded, we weld its vertices and perform an in-place vertex cache optimimzation. This performs the attribute sort as well.

```
        // Attempt to optimize the mesh
        pNewMesh->WeldVertices( 0 );
        pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );
```

We call the CScene::AddMesh function to make room for the new mesh in the scene mesh array and add the new mesh pointer to the end of the list.

```
        // Store this new mesh
        if ( AddMesh( ) < 0 ) { delete pNewMesh; return false; }
        m_pMesh[ m_nMeshCount - 1 ] = pNewMesh;

        // Store as object reference mesh
        pReferenceMesh = pNewMesh;

    } // End if mesh doesn't exist.
```

To position the mesh in the scene, we create a new CObject and store the mesh pointer and the entity world matrix inside the newly allocated CObject structure. The object is added to our global list and the function is complete. The CScene::AddObject function is very much like the CScene::AddMesh function in that it resizes the scene's CObject array making room for a new entry at the end.

```
    // Now build an object for this mesh (standard identity)
    CObject * pNewObject = new CObject( pReferenceMesh );
    if ( !pNewObject ) return false;

    // Copy over the specified matrix
    pNewObject->m_mtxWorld = mtxWorld;

    // Store this object
    if ( AddObject() < 0 ) { delete pNewObject; return false; }
    m_pObject[ m_nObjectCount - 1 ] = pNewObject;

    // Success!!
    return true;
}
```

CScene::ProcessSkyBox

ProcessSkyBox (called by CScene::ProcessEntities) adds six cube faces (12 triangles) to the scene skybox mesh and loads the six textures whose names are stored in the skybox entity passed into the function. This function provides some useful insight into populating a CTriMesh manually. It also demonstrates creation of a managed-mode mesh.

When we wish to create a managed-mode mesh from X file data, we would normally register the CollectTexture callback with the mesh so that the mesh loading function can pass the texture filenames found in the X file to the function and get back texture pointers for its own internal storage. The

managed mesh stores the texture pointer along with the matching material for each of its subsets in its internal attribute data array. In this particular case (skybox) we will not be loading the data from an X file, so we will not need to register the callback. We will manually add the texture and material information for each mesh subset to the mesh's internal attribute array ourselves.

The first thing we need to do is call CTriMesh::SetDataFormat to tell the mesh about our desired vertex/index formats. Because we will be adding vertices and indices to the cube mesh one face at a time (i.e. two triangles at a time) each cube face will consist of 4 vertices and 6 indices. We will use two temporary arrays, an index array and a vertex array, to build the cube faces as we go along. The front face case is examined first.

```
bool CScene::ProcessSkyBox( const SkyBoxEntity & SkyBox )
{
    MESH_ATTRIB_DATA * pAttribData;
    USHORT             Indices[6];
    CVertex            Vertices[4];

    // Set the mesh data format
    m_SkyBoxMesh.SetDataFormat( VERTEX_FVF, sizeof(USHORT) );

    // Build Front quad (remember all quads point inward)
    Vertices[0] = CVertex(-10.0f, 10.0f, 10.0f, D3DXVECTOR3(0.0f, 0.0f, 0.0f), 0.0f, 0.0f);
    Vertices[1] = CVertex( 10.0f, 10.0f, 10.0f, D3DXVECTOR3(0.0f, 0.0f, 0.0f), 1.0f, 0.0f);
    Vertices[2] = CVertex( 10.0f,-10.0f, 10.0f, D3DXVECTOR3(0.0f, 0.0f, 0.0f), 1.0f, 1.0f);
    Vertices[3] = CVertex(-10.0f,-10.0f, 10.0f, D3DXVECTOR3(0.0f, 0.0f, 0.0f), 0.0f, 1.0f);

    // Build the skybox indices
    Indices[0] = 0; Indices[1] = 1; Indices[2] = 3;
    Indices[3] = 1; Indices[4] = 2; Indices[5] = 3;
```

To add the indices and vertices for the quads to the mesh we call the AddVertex and AddFace functions. Notice that we are adding 2 triangles for the face and that we pass in an attribute ID of 0. Since each quad will have a different texture mapped to it, each will belong to a different subset. We will incremement this attribute ID for each quad that we add, such that the front face has an attribute ID of 0, the back face has an attribute ID of 1, and so on. The resulting mesh of the cube will contain 12 triangles and six subsets. Each subset represents one cube face and contains two triangles.

```
    // Add the vertices and indices to this mesh
    m_SkyBoxMesh.AddVertex( 4, &Vertices );
    m_SkyBoxMesh.AddFace( 2, Indices, 0 );
```

We repeat the process for the remaining five faces of the cube. Keep in mind when examining the vertex data that these cube faces are inward facing, so the vertex winding order will reflect this fact. Note as well that the zero length normal may seem strange, but the skybox will be rendered with lighting disabled anyway so the normal information will not be used. This allows us to use the same vertex flags for the skybox as the rest of the objects in our scene. Thus, we avoid calling IDirect3DDevice9::SetFVF in the scene's main render function to change vertex formats every time we render the skybox. Alternatively, you can use a vertex format with position and UV coordinates only if you prefer.

```
    // Back Quad
    Vertices[0] = CVertex( 10.0f, 10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 0.0f );
    Vertices[1] = CVertex(-10.0f, 10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 0.0f );
    Vertices[2] = CVertex(-10.0f,-10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 1.0f );
    Vertices[3] = CVertex( 10.0f,-10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 1.0f );

    // Build the skybox indices
    Indices[0] = 4; Indices[1] = 5; Indices[2] = 7;
    Indices[3] = 5; Indices[4] = 6; Indices[5] = 7;

    // Add the vertices and indices to this mesh
    m_SkyBoxMesh.AddVertex( 4, &Vertices );
    m_SkyBoxMesh.AddFace( 2, Indices, 1 );

    // Left Quad
    Vertices[0] = CVertex(-10.0f, 10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 0.0f );
    Vertices[1] = CVertex(-10.0f, 10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 0.0f );
    Vertices[2] = CVertex(-10.0f,-10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 1.0f );
    Vertices[3] = CVertex(-10.0f,-10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 1.0f );

    // Build the skybox indices
    Indices[0] = 8; Indices[1] = 9; Indices[2] = 11;
    Indices[3] = 9; Indices[4] = 10; Indices[5] = 11;

    // Add the vertices and indices to this mesh
    m_SkyBoxMesh.AddVertex( 4, &Vertices );
    m_SkyBoxMesh.AddFace( 2, Indices, 2 );

    // Right Quad
    Vertices[0] = CVertex( 10.0f, 10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 0.0f );
    Vertices[1] = CVertex( 10.0f, 10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 0.0f );
    Vertices[2] = CVertex( 10.0f,-10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 1.0f );
    Vertices[3] = CVertex( 10.0f,-10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 1.0f );

    // Build the skybox indices
    Indices[0] = 12; Indices[1] = 13; Indices[2] = 15;
    Indices[3] = 13; Indices[4] = 14; Indices[5] = 15;

    // Add the vertices and indices to this mesh
    m_SkyBoxMesh.AddVertex( 4, &Vertices );
    m_SkyBoxMesh.AddFace( 2, Indices, 3 );

    // Top Quad
    Vertices[0] = CVertex(-10.0f, 10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 0.0f );
    Vertices[1] = CVertex( 10.0f, 10.0f -10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 0.0f );
    Vertices[2] = CVertex( 10.0f, 10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 1.0f );
    Vertices[3] = CVertex(-10.0f, 10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 1.0f );

    // Build the skybox indices
    Indices[0] = 16; Indices[1] = 17; Indices[2] = 19;
    Indices[3] = 17; Indices[4] = 18; Indices[5] = 19;

    // Add the vertices and indices to this mesh
    m_SkyBoxMesh.AddVertex( 4, &Vertices );
    m_SkyBoxMesh.AddFace( 2, Indices, 4 );
    // Bottom Quad
    Vertices[0] = CVertex(-10.0f,-10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 0.0f );
    Vertices[1] = CVertex( 10.0f,-10.0f, 10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 0.0f );
    Vertices[2] = CVertex( 10.0f,-10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 1.0f, 1.0f );
```

```
    Vertices[3] = CVertex(-10.0f,-10.0f,-10.0f, D3DXVECTOR3( 0.0f, 0.0f, 0.0f), 0.0f, 1.0f );

    // Build the skybox indices
    Indices[0] = 20; Indices[1] = 21; Indices[2] = 23;
    Indices[3] = 21; Indices[4] = 22; Indices[5] = 23;

    // Add the vertices and indices to this mesh
    m_SkyBoxMesh.AddVertex( 4, &Vertices );
    m_SkyBoxMesh.AddFace( 2, Indices, 5 );
```

At this point the skybox mesh contains the six quads (12 triangles) needed and has an attribute buffer with values ranging from 0 to 5. Thus we have six subsets, with two triangles per subset.

To make the mesh self-contained, we populate its internal attribute data array with the appropriate texture and material for each subset. We allocate space in the mesh MESH_ATTRIB_DATA array for 6 elements, one for each subset, by calling AddAttributeData. We follow this with a call to GetAttributeData to return a MESH_ATTRIB_DATA pointer to the first element in the array. We will then populate the array with the appropriate subset rendering resources. Note that the skybox will use the same default white material for each subset. CScene::CollectTexture is used to load and/or retrieve the texture pointer based on the filenames stored in the skybox entity. Thus, these skybox textures will exist in the scene's main texture array like all the others.

```
    // Add the attribute data (We'll let the skybox manage itself)
    m_SkyBoxMesh.AddAttributeData( 6 );
    pAttribData = m_SkyBoxMesh.GetAttributeData();

    D3DMATERIAL9 Material;
    ZeroMemory( &Material, sizeof(D3DMATERIAL9));
    Material.Diffuse = D3DXCOLOR( 1.0, 1.0, 1.0, 1.0f );
    Material.Ambient = D3DXCOLOR( 1.0, 1.0, 1.0, 1.0f );

    // Build and set the attribute data for the skybox
    for ( ULONG i = 0; i < 6; ++i )
    {
        pAttribData[i].Texture  = CollectTexture( this, SkyBox.Textures[i] );
        pAttribData[i].Material = Material;
        if ( pAttribData[i].Texture ) pAttribData[i].Texture->AddRef();
    } // Next Texture
```

We also remember to call AddRef for each texture interface since we are making a copy of the pointer.

At this point the mesh contains all subset attribute information as well as the vertex and index data (in its temporary system memory vertex and index arrays). The underlying skybox ID3DXMesh has not yet been created, so this is our final step before returning to the caller.

```
    // Build the mesh
    m_SkyBoxMesh.BuildMesh( D3DXMESH_MANAGED, m_pD3DDevice );

    return true;
}
```

We have now discussed all functions of significance with respect to loading the IWF file and populating the scene's mesh, object and resource arrays. As it stands, the scene has everything it needs to render. Let us now look at how the CScene::Render function (called by CGameApp::FrameAdvance) renders all of its meshes.

CScene::Render

The CScene::Render function is responsible for rendering all the meshes which comprise the scene. In this application, all meshes, with the exception of one, are being used in non-managed mode. So this function has to be responsible for setting the states before rendering the subsets of each mesh. The exception is the scene's skybox mesh which is utilizing managed mode. The CScene::RenderSkyBox function is called to set the skybox's position in the world and render it. We will study the RenderSkyBox function shortly.

Depending on the type of scene being rendered, it may be more or less efficient to batch render based on either attribute or transform. We dicussed the various pros and cons of each approach earlier. To allow our code to easily be adjusted to use both rendering strategies, a pre-compiler define is used to control how this function is compiled. Using this technique, we can actually instruct the compiler, as our code is processed, to include or exclude certain sections of the code from the final build.

If we define DRAW_ATTRIBUTE_ORDER with a value of 1, the code that batch renders the scene based on attributes will be compiled. This will render the entire scene based on subsets minimizing texture and material state changes. This would be a suitable strategy if many of your meshes exist in world space and do not need to be transformed.

If we define DRAW_ATTRIBUTE_ORDER with a value of 0, then an alternative version of the render code will be compiled. This time the code will batch render on a per object basis, minimizing transform state changes.

In this demonstration we set DRAW_ATTRIBUTE_ORDER to 0 by default so that it renders on a per object basis by default. Feel free to change this value to 1 so that you can benchmark the different rendering strategies with different scenes.

The code will be discussed a section at a time. This function is longer than would have been the case had a single rendering strategy been used, but it is still pretty straightforward. Which of the two versions of the rendering code actually gets compiled is controlled by the directive just mentioned.

```
void CScene::Render( CCamera & Camera )
{
    ULONG i, j;
    long  MaterialIndex, TextureIndex;

    if ( !m_pD3DDevice ) return;

    // Setup some states...
```

```
    m_pD3DDevice->SetRenderState( D3DRS_NORMALIZENORMALS, TRUE );

    // Render the skybox first !
    RenderSkyBox( Camera );
```

The first thing the code does is set the D3DRS_NORMALIZENORMALS render state to true. This render state forces the pipeline to ensure that vertex normals are unit length before being used for the lighting calculations in the DirectX pipeline. While the normals of a mesh would almost certainly be unit length to begin with, using a world matrix that scales vertices can cause the normals of the vertex to be scaled when the vertex is transformed by the pipeline. If such a scaling matrix is being used, the vertex normals will no longer be unit length, resulting in incorrect lighting calculations. This render state addresses that potential problem.

The first thing we render is the skybox mesh by issuing a call to CScene::RenderSkyBox. The skybox must be rendered first so that its faces are rendered behind all other scene objects (making the skybox scenery appear distant).

The next section of code activates the lights used by the scene. Placing this code here makes sure that if the user loads a new scene with more (or fewer) lights than are currently active, they will automatically be set the next time the scene is rendered. This also makes sure that if the device is reset, the lights are also automatically reset. This code is placed here for simplicity and would be moved outside the main render loop in a real world situation (light states should not be needlessly set every time a frame is rendered).

```
    // Enable lights
    for ( i = 0; i < m_nLightCount; ++i )
    {
        m_pD3DDevice->SetLight( i, &m_pLightList[i] );
        m_pD3DDevice->LightEnable( i, TRUE );
    }
```

Next we define DRAW_ATTRIBUTE_ORDER to 0 so that the scene is rendered per object instead of per subset. By changing this value to 1, you can recompile the code and force the alternative rendering strategy to be used instead. It should be noted that both rendering strategies are mutually exclusive. That is, only one of the possible two sections of rendering code can be compiled into the application at any given time.

```
// For this demo
#define DRAW_ATTRIBUTE_ORDER 0
```

If we have not set DRAW_ATTRIBUTE_ORDER to 0 then it means we wish to compile the code that batch renders the scene across mesh boundries. This is done by looping through every attribute in the scene global list and rendering any meshes that contain a subset with a matching global attribute ID. This minimimzes texture changes but increases the number of times we must set the world matrix. As we are looping through the scene attributes in the outer loop and objects in the inner loop, the world matrix for a particular mesh must be set many times, once for each subset.

The code loops through each of the scene attributes and extracts from the attribute array the texture index and the material index used by that attribute. It uses the material index to get a pointer to the correct material in the scene's material array and sets it as the current material on the device. If the current scene attribute does not contain a material, then we set a default material. We also use the retrieved texture index to bind the correct texture to texture stage 0. If the current scene attribute contains no texture, we set texture stage 0 to NULL.

```
#if ( DRAW_ATTRIBUTE_ORDER != 0 )

    // Loop through each scene owned attribute
    for ( j = 0; j < m_nAttribCount; j++ )
    {
        // Retrieve indices
        MaterialIndex = m_pAttribCombo[j].MaterialIndex;
        TextureIndex  = m_pAttribCombo[j].TextureIndex;

        // Set the states
        if ( MaterialIndex >= 0 )
            m_pD3DDevice->SetMaterial( &m_pMaterialList[ MaterialIndex ] );
        else
            m_pD3DDevice->SetMaterial( &m_DefaultMaterial );

        if ( TextureIndex >= 0 )
            m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ]->Texture );
        else
            m_pD3DDevice->SetTexture( 0, NULL );
```

Now that we have the texture and the material set for this attribute, we will loop through every object in the scene, get a pointer to its CTriMesh object, set its world matrix and FVF flags, and draw the current subset. Of course, the subset we are currently processing may not exist in the mesh and it will result in a no-op and quickly return from the function.

```
        // Process each object
        for ( i = 0; i < m_nObjectCount; ++i )
        {
            CD3DXMesh * pMesh = m_pObject[i]->m_pMesh;
            if ( !pMesh ) continue;

            // Setup the per-mesh / object details
            m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i]->m_mtxWorld );
            m_pD3DDevice->SetFVF( pMesh->GetFVF() );

            // Render all faces with this attribute ID
            pMesh->DrawSubset( j );

        } // Next Object

    } // Next Attribute
```

The above code repeats for every scene attribute until all subsets of all non-managed meshes have been rendered.

If DRAW_ATTRIBUTE_ORDER is set to 0, alterative rendering code will be compiled instead. This code will render on a per object basis (which proved to be much faster in our example application). This is because it minimizes potential FVF changes and the number of times we have to set the world matrix, thus reducing stalls in the pipeline.

This approach (really just a re-ordering of the code shown above), loops through each of the scene objects and sets its FVF flags and associated world matrix. The next step loops through each of the scenes attributes, setting the material and texture for that attribute. Finally, all meshes are instructed to render any subsets which have matching subset IDs.

```
#else

    // Process each object
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CD3DXMesh * pMesh = m_pObject[i]->m_pMesh;
        if ( !pMesh ) continue;

        // Setup the per-mesh / object details
        m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i]->m_mtxWorld );
        m_pD3DDevice->SetFVF( pMesh->GetFVF() );

        // Loop through each scene owned attribute
        for ( j = 0; j < m_nAttribCount; j++ )
        {
            // Retrieve indices
            MaterialIndex = m_pAttribCombo[j].MaterialIndex;
            TextureIndex  = m_pAttribCombo[j].TextureIndex;

            // Set the states
            if ( MaterialIndex >= 0 )
                m_pD3DDevice->SetMaterial( &m_pMaterialList[ MaterialIndex ] );
            else
                m_pD3DDevice->SetMaterial( &m_DefaultMaterial );

            if ( TextureIndex >= 0 )
                m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ]->Texture );
            else
                m_pD3DDevice->SetTexture( 0, NULL );

            // Render all faces with this attribute ID
            pMesh->DrawSubset( j );

        } // Next Object

    } // Next Attribute

#endif // !DRAW_ATTRIBUTE_ORDER != 0
```

Before exiting, we disable any currently active lights so that if we load another IWF scene which uses fewer lights than the current scene, the current scene's lights do not remain active and influence the new scene. Of course, this could be done outside the render loop, and certainly would be in a real world situation. We placed it here for simplicity in our simple application.

```
    // Disable lights again (to prevent problems later)
    for ( i = 0; i < m_nLightCount; ++i ) m_pD3DDevice->LightEnable( i, FALSE );

}
```

CScene::RenderSkyBox

The CScene::RenderSkyBox function builds a world matrix for the skybox that will translate it to the camera's current position. This ensures that the camera remains at the center of the box at all times. We simply place the camera position in the fourth row (the translation vector) of the world matrix and render the cube.

```
void CScene::RenderSkyBox( CCamera & Camera )
{
    // Bail if there is no sky box
    if ( m_SkyBoxMesh.GetNumFaces() == 0 ) return;

    D3DXMATRIX mtxWorld, mtxIdentity;
    D3DXMatrixIdentity( &mtxWorld );
    D3DXMatrixIdentity( &mtxIdentity );

    // Generate our sky box rendering origin and set as world matrix
    D3DXVECTOR3 CamPos = Camera.GetPosition();
    D3DXMatrixTranslation( &mtxWorld, CamPos.x, CamPos.y + 1.3f, CamPos.z );
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxWorld );
```

We disable lighting and depth buffering since neither is appropriate for this type of effect. Lighting would create visible seams and strange coloring. Depth buffering is both unnecessary and undesirable -- it is unnecessary because we do not want to perform thousands of per-pixel depth tests when we know that the depth buffer is currently clear, and it is undesirable because we do not want to write depth values that may potentially occlude other scene objects (the skybox is only a background and should never occlude anything).

```
    // Set up rendering states for the sky box
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_FALSE );
```

When rendering a skybox, we set the texture addressing modes for both the U and V axes to D3DTADDRESS_CLAMP. This prevents pixels on the edge of each quad from being bilinearly filtered with pixels on the opposite side of the texture, creating a visible seam.

```
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_ADDRESSU, D3DTADDRESS_CLAMP );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_ADDRESSV, D3DTADDRESS_CLAMP );
```

Since the skybox is a managed mesh, it will handle setting all texture and render states. It contains the textures used by each of its faces in its internal attribute array which we populated ealier. We simply need to call the self-contained CTriMesh::Draw function.

```
    // Render the sky box
    m_SkyBoxMesh.Draw( );
```

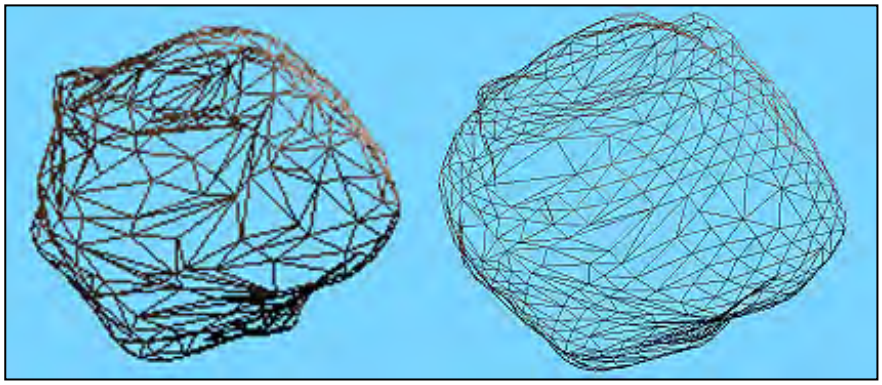
Finally, we reset all render and sampler states and the device world matrix before we return so that we do not influence how the rest of the scene is rendered.

```
    // Reset our states
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_ADDRESSU, D3DTADDRESS_WRAP );
    m_pD3DDevice->SetSamplerState( 0, D3DSAMP_ADDRESSV, D3DTADDRESS_WRAP );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, TRUE );
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxIdentity );
}
```

Lab Project 8.2: Progressive Meshes

This lab project will add Progressive Mesh support to the code we studied in Lab Project 8.1.

To accomplish our objective, we will slightly modify our CTriMesh class so that it has the ability to be used as a static or progressive mesh. Most of the functionality for the mesh class (and indeed the entire application) will be similar to the code studied in the last project.



The CTriMesh (see CObject.h) object will now store two pointers: an ID3DXMesh pointer and an ID3DXPMesh pointer. If the m_pPMesh pointer is active then the progressive mesh will be used by all drawing and optimization functions. If not, the standard mesh will be used. This allows us to easily use the same mesh for both regular and progressive meshes and change from one to the other easily within a CTriMesh object.

```
class CTriMesh
{
  …

  LPD3DXMESH          m_pMesh;                        // Physical mesh object
  LPD3DXPMESH         m_pPMesh;                       // Physical PMesh object
```

We will add a few wrapper functions to set the current level of detail for the underlying progressive mesh. CTriMesh::SetNumVertices and CTriMesh::SetNumFaces simply pass on the request to the underlying ID3DXPMesh if it exists. They do nothing if the progressive mesh has not been generated.

```
  HRESULT           SetNumFaces       ( ULONG FaceCount );
  HRESULT           SetNumVertices    ( ULONG VertexCount );
```

We also include two more wrapper functions for trimming the base level of detail of the underlying progressve mesh. Once again, these functions do nothing if the progressive mesh has not been generated.

```
  HRESULT           TrimByFaces       ( ULONG NewFacesMin, ULONG NewFacesMax );
  HRESULT           TrimByVertices    ( ULONG NewVerticesMin, ULONG NewVerticesMax );
```

CTriMesh also provides a function to generate its progressive mesh using its underlying ID3DXMesh as the input for progressive mesh generation. This call essentially places the CTriMesh object into progressive mesh mode. The underlying ID3DXMesh interface must have been created before switching

to progressive mode. This is because the regular ID3DXMesh is used as the input mesh for D3DX progressive mesh generation.

```
 HRESULT          GeneratePMesh( CONST LPD3DXATTRIBUTEWEIGHTS pAttributeWeights,
                                 CONST FLOAT *pVertexWeights, ULONG MinValue,
                                 ULONG Options, bool ReleaseOriginal = true );
```

Cloning and optimizing are also supported for both standard and progressive meshes.

```
HRESULT          CloneMeshFVF ( ULONG Options, ULONG FVF, CTriMesh * pMeshOut,
                                  MESH_TYPE MeshType = MESH_DEFAULT,
                                  LPDIRECT3DDEVICE9 pD3DDevice = NULL );

HRESULT          Optimize     ( ULONG Flags, CTriMesh * pMeshOut,
                                  MESH_TYPE MeshType = MESH_DEFAULT,
                                  LPD3DXBUFFER * ppFaceRemap = NULL,
                                  LPD3DXBUFFER * ppVertexRemap = NULL,
                                  LPDIRECT3DDEVICE9 pD3DDevice = NULL );
```

Notice that we use a new parameter in both function calls. This is a member of the MESH_TYPE enumerated type. By default, this parameter is set to MESH_DEFAULT, which means that the output mesh will be the same type as the source mesh being cloned. We can also specify standard or progressive flags to convert between one mesh type and the other. This allows us, for example, to take a CTriMesh in progressive mesh mode and clone from another CTriMesh object in regular mesh mode and vice versa.

When cloning a CTriMesh in regular mesh mode out to a CTriMesh in progressive mesh mode, the underlying progressive mesh in the cloned object will use the regular mesh in the source object as its base (highest level of detail) geometry. When cloning a CTriMesh in progressive mesh mode to a CTriMesh in regular mesh mode, the underlying ID3DXMesh in the cloned object will contain the geometry taken from the progressive mesh of the source object at its *current* level of detail. This allows us to create a progressive CTriMesh object and simplify the data by altering its current detail setting, before cloning it back out to a new CTriMesh in regular mesh mode. The source mesh can then be released, leaving us with a simplified standard CTriMesh that can be rendered without the runtime overhead a progressive mesh would incur. If we need the ability to alter the geometric detail of a CTriMesh at runtime, we will want to use a CTriMesh in progressive mesh mode for rendering.

MESH_TYPE is now part of the CTriMesh namespace and is defined in CObject.h as:

```
enum MESH_TYPE     { MESH_DEFAULT = 0, MESH_STANDARD = 1, MESH_PROGRESSIVE = 2 };
```

Finally, CTriMesh now includes some utility wrapper functions for working with the progressive mesh and extracting its current settings. These functions should be self explanatory given our discussion of ID3DXPMesh in the textbook.

```
ULONG            GetMaxFaces       ( ) const;
ULONG            GetMaxVertices    ( ) const;
```

```
ULONG             GetMinFaces       ( ) const;
ULONG             GetMinVertices    ( ) const;
```

If the CTriMesh is not in progressive mesh mode, then all of these functions will return the number of vertices/faces in the regular ID3DXMesh. The existence of an underlying ID3DXPMesh interface takes precedence in such functions, in which case the functions return information about the progressive mesh.

Our application can get a pointer to the underlying ID3DXPMesh interface by calling the GetPMesh function. If the progressive mesh has not been generated, this call will return NULL.

```
LPD3DXPMESH       GetPMesh          ( ) const;
```

While the next two functions are not new to the CTriMesh class, they now behave differently when the mesh object is in progressive mode. In standard mode, these functions return the number of vertices and faces in the temporary arrays used to manually generate the ID3DXMesh when the CTriMesh::BuildMesh function is called. If the ID3DXMesh has already been built, then they return the number of vertices and faces in the underlying D3DXMesh object. But if the mesh object has its progressive mesh generated, then these functions will return the number of vertices and faces used to render the progressive mesh at its current level of detail.

```
ULONG             GetNumVertices    ( ) const;
ULONG             GetNumFaces       ( ) const;
```

The final new function we will add is called SnapshotToMesh. It clones the CTriMesh progressive mesh out to a standard mesh and stores the result in the CTriMesh ID3DXMesh pointer. This call does not destroy the current progressive mesh or change the mode of the CTriMesh object from progressive mesh to standard mesh mode unless the ReleaseProgressive Booelan parameter is set to true. This is handy because it allows us to create a CTriMesh object, generate a progressive mesh for it, lower its level of detail, and clone the current LOD progressive mesh to the underlying standard one. We can then release the progressive mesh, to place the same CTriMesh back into standard mode. We have simply used the progressive mesh in this case to perform a one-time simplification procedure on the mesh data.

```
HRESULT  SnapshotToMesh ( bool ReleaseProgressive = true );
```

If we specify false as the input parameter then the underlying progressive mesh will not be released and the CTriMesh remains in progressive mode. This is also useful since we can attach the simplified mesh to a new CTriMesh if desired. This allows for easy cloning of a progressive mode CTriMesh to a standard CTriMesh. In fact, we use this mechanism in CloneMeshFVF and Optimize to do exactly that.

Before we start examining the underlying code for these new and modified functions in more detail, let us briefly look at some usage scenarios for the updated CTriMesh.

Using CTriMesh to Create a Progressive Mesh

Creating a progressive CTriMesh is a two step process. Step one is exactly the same as our mesh creation code in the last project -- we simply build the mesh so that the underlying ID3DXMesh is created. Step two involves calling CTriMesh::GeneratePMesh to generate the ID3DXPMesh from the underlying ID3DXMesh. This places the CTriMesh in progressive mesh mode and will (by default) release the ID3DXMesh interface. The newly generated ID3DXPMesh will now be used for all rendering. The following example code shows how we might create a progressive CTriMesh using data loaded in from an X file stored on disk:

```
CTriMesh * pNewMesh = new CTriMesh;
pNewMesh->RegisterCallback( CTriMesh::CALLBACK_ATTRIBUTEID, CollectAttributeID, this );
pNewMesh->LoadMeshFromX( 'MyXFile.x', D3DXMESH_MANAGED, m_pD3DDevice );
pNewMesh->WeldVertices( 0 );
pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );
```

At this point we have an optimized ID3DXMesh. So far, this has been exactly like the code in Lab Project 8.1. Now we call CTriMesh::GeneratePMesh to place the CTriMesh object into progressive mesh mode. We will pass in NULL in this example for the first two parameters so the default vertex component weighting is used (all vertices are assumed to have the same weight, and therefore will all be assigned the same priority for collapse). We also specify that the underlying progressive mesh should be generated such that it stores enough edge collapse structures to allow us to dynamically simplify down to a face count of 50. This target might not be possible, but the progressive mesh will calculate enough collapse structures to get as close to this number as possible without harming the topology of the mesh to an unacceptable degree.

```
pNewMesh->GeneratePMesh( NULL, NULL, 50, D3DXMESHSIMP_FACE );
```

At this point, the CTriMesh has released its ID3DXMesh interface and now has its m_pPMesh pointer pointing to a valid ID3DXPMesh interface. Finally, we optimize the progressive mesh for rendering.

```
pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );
```

CTriMesh::GeneratePMesh

This function will use the standard mesh (CTriMesh::m_pMesh) as the base level of detail for a new progressive mesh. Therefore, the standard mesh should already be a valid ID3DXMesh when this function is called.

```
HRESULT CTriMesh::GeneratePMesh( CONST LPD3DXATTRIBUTEWEIGHTS pAttributeWeights,
                                 CONST FLOAT *pVertexWeights, ULONG MinValue,
                                 ULONG Options, bool ReleaseOriginal /* = true */ )
{
    HRESULT         hRet;
    LPD3DXMESH      pTempMesh      = NULL;
    LPD3DXBUFFER    pTempAdjacency = NULL;

    // Validate Parameters
```

```
    if (!m_pMesh) return D3DERR_INVALIDCALL;
```

The first thing the function does is ensure that m_pMesh is not NULL. If it is, then the function has been called prematurely and we return immediately. If the standard mesh has been created, then we release the current progressive mesh to make room for the new one we are about to create.

```
    // Release previous Progressive mesh if one already exists
    if ( m_pPMesh ) { m_pPMesh->Release(); m_pPMesh = NULL; }
```

To create a progressive mesh, we need adjacency information for the D3DXGeneratePMesh function call. If we have not created the adjacency array previously, we do so now.

```
    // Generate adjacency if not yet available
    if (!m_pAdjacency)
    {
        GenerateAdjacency();
    }
```

Next we need to validate the standard mesh to make sure it does not contain invalid data that would cause the generation of a progressive mesh to fail. We call D3DXValidMesh and pass in a pointer to the standard mesh along with its adjacency data.

```
    // Validate the base mesh
    hRet = D3DXValidMesh( m_pMesh, (DWORD*)m_pAdjacency->GetBufferPointer(), NULL );
```

If the D3DXValidMesh function fails, then the standard mesh contains invalid geometry. We will then attempt to repair the damage using the D3DXCleanMesh function. D3DXCleanMesh does not alter the actual geometry of the mesh being cleaned -- instead it clones the clean data out to a new ID3DXMesh. It also fills a new adjacency buffer, so we must allocate this buffer and pass its pointer into the function.

```
   if ( FAILED(hRet) )
   {
        // Allocate the temporary adjacency buffer
        hRet = D3DXCreateBuffer( m_pAdjacency->GetBufferSize(), &pTempAdjacency );
        if ( FAILED(hRet) ) {  return hRet; }

        // Clean the mesh data storing the new mesh interface pointer in pTempMesh.
        hRet = D3DXCleanMesh( m_pMesh, (DWORD*)m_pAdjacency->GetBufferPointer(), &pTempMesh,
                              (DWORD*)pTempAdjacency->GetBufferPointer(), NULL );

        if ( FAILED(hRet) ) { pTempAdjacency->Release(); return hRet; }
    }
```

The clean mesh is assigned to the local ID3DXMesh pointer pTempMesh. If the function fails, then there is nothing we can do; the data is simply incompatible with progressive mesh generation. If this is the case, we release the temporary adjacency buffer and return.

If the function is successful, pTempMesh will point to the cleaned ID3DXMesh interface pointer and pTempAdjacency will contain the updated face adjacency information.

If m_pMesh was valid to begin with, then we copy the mesh interface pointer and its adjacency buffer into the pTempMesh and pTempAdjacency pointers. This way, whether the mesh needed to be cleaned or not, pTempAdjacency and pTempMesh point to the adjacency buffer and the mesh we will use to create the progressive mesh.

```
    else
    {
        // Simply store common pointers
        pTempAdjacency = m_pAdjacency;
        pTempMesh      = m_pMesh;

        // Add references so that the originals are not released
        m_pAdjacency->AddRef();
        m_pMesh->AddRef();

    } // End if valid mesh
```

We generate the progressive mesh with a call to D3DXGeneratePMesh, passing in the source mesh and other required parameters. We store the new progressive mesh in the CTriMesh::m_pPMesh member pointer when complete and the CTriMesh object is now in progressive mesh mode.

```
  // Generate the progressive mesh
  hRet = D3DXGeneratePMesh(pTempMesh, (DWORD*)pTempAdjacency->GetBufferPointer(),
                           pAttributeWeights, pVertexWeights, MinValue, Options, &m_pPMesh );
```

We can now release pTempMesh and pTempAdjacency since we no longer need them.

```
    // Release all used objects
    pTempMesh->Release();
    pTempAdjacency->Release();
```

Finally, if the ReleaseOriginal Boolean paremeter is set to true (the default) then we release the standard mesh interface (m_pMesh) as well. From this point forward, any member functions we call will work with the underlying progressive mesh instead. Note that we only release the standard mesh if the progressive mesh was successfully generated.

```
    // Release the original mesh if requested, and PMesh generation was a success
    if ( ReleaseOriginal && SUCCEEDED(hRet) ) { m_pMesh->Release(); m_pMesh = NULL; }

    // Success??
    return hRet;
}
```

CTriMesh::CloneMeshFVF

The CloneMeshFVF function has had a fair bit of code added to it to cope with the dual-use nature of the modified CTriMesh class. The MESH_TYPE parameter allows the caller to specify whether the clone will be a progressive mesh (MESH_PROGRESSIVE), a standard mesh (MESH_STANDARD), or the same type as the source mesh (MESH_DEFAULT).

The results differ based on the source mesh type and the requested destination type. The semantics are detailed below:

Source Mesh: MESH_STANDARD
Destination Mesh: MESH_STANDARD -or- MESH_DEFAULT
In this case the output CTriMesh will be a standard ID3DXMesh. The vertices, faces, and attributes will be copied into the output mesh creating a duplicate CTriMesh. Vertex/index format may change depending on the Options and FVF flags passed into the function.

Source Mesh: MESH_STANDARD
Destination Mesh: MESH_PROGRESSIVE
In this case the output CTriMesh will be an ID3DXPMesh. The source mesh determines the base LOD for the progressive mesh. The source mesh remains in standard mode and the output mesh will be in progressive mesh mode. Vertex/index format may change depending on the Options and FVF flags passed into the function.

Source Mesh: MESH_PROGRESSIVE
Destination Mesh: MESH_PROGRESSIVE -or- MESH_DEFAULT
In this case the output CTriMesh will be an ID3DXPMesh. The source progressive mesh is copied in its entirety and the output mesh will have the same face and vertex count as the source. Vertex/index format may change depending on the Options and FVF flags passed into the function.

Source Mesh: MESH_PROGRESSIVE
Destination Mesh: MESH_STANDARD
In this case the output CTriMesh will be an ID3DXMesh. The clone will contain only geometry that is being used at the current level of detail for the source progressive mesh. This is essentially a snapshot of the source mesh at its current LOD. Vertex/index format may change depending on the Options and FVF flags passed into the function.

The first thing the clone function does is make sure that the mesh about to be cloned has a valid source mesh (standard or progressive). If the progressive mesh exists, then the mesh is assumed to be in progressive mesh mode and the standard mesh will be ignored. The progressive mesh will therefore be used as the source mesh in the cloning operation. We also generate face adjacency for the source mesh if it has not been previously generated because it is needed for the cloning operation. Note that the final parameter to this function is a pointer to an IDirect3DDevice9 interface describing the device we would like the output mesh to belong to. Usually we will want this to be the same as the source mesh that is

about to be cloned. If so, we can set this parameter to NULL and the device retrieved from the current mesh will be used.

```
HRESULT CTriMesh::CloneMeshFVF(ULONG Options, ULONG FVF, CTriMesh * pMeshOut,
                               MESH_TYPE MeshType , LPDIRECT3DDEVICE9 pD3DDevice)
{
    HRESULT         hRet;
    LPD3DXBASEMESH  pCloneMesh  = NULL;

    // Validate requirements
    if ( (!m_pMesh && !m_pPMesh) || !pMeshOut ) return D3DERR_INVALIDCALL;

    // Generate adjacency if not yet available
    if (!m_pAdjacency)
    {
        hRet = GenerateAdjacency();
        if ( FAILED(hRet) ) return hRet;

    } // End if no adjacency

    // If no new device was passed...
    if ( !pD3DDevice )
    {
        // we'll use the same device as this mesh
        // This automatically calls 'AddRef' for the device
        pD3DDevice = GetDevice();
    }
    else
    {
        // Otherwise we'll add a reference here so that we can
        // release later for both cases without doing damage :)
        pD3DDevice->AddRef();
    }
```

The next step is to determine the requested destination mesh type and process the request as described above.

If MESH_DEFAULT was passed in then we create a cloned CTriMesh of the same type as the current mesh object being cloned. Progressive meshes take priority, so if the m_pPMesh pointer is not NULL then the mesh is currently in progressive mesh mode and we will need to clone a new progressive mesh using the D3DXClonePMeshFVF function. If the m_pPMesh pointer is NULL then we assume the mesh is in standard mode and clone the standard mesh using D3DXCloneMeshFVF. Note that we store the cloned interface as an ID3DXBaseMesh. This allows us to use the same pointer type for both types.

```
    switch ( MeshType )
    {
        case MESH_DEFAULT:  // Continue to use the same type as 'this'
            if ( m_pPMesh )
            {
                // Attempt to clone the mesh
                hRet = m_pPMesh->ClonePMeshFVF( Options, FVF, pD3DDevice, \
                                                (LPD3DXPMESH*)&pCloneMesh );
                if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }
```

```
            } // End if Progressive
            else
            {
                // Attempt to clone the mesh
                hRet = m_pMesh->CloneMeshFVF( Options, FVF, pD3DDevice, \
                                              (LPD3DXMESH*)&pCloneMesh );
                if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }

            } // End if standard
            break;
```

Processing the request for a standard output mesh is just as easy. Again, we start by checking our progressive mesh first and then fall back to the standard mesh if necessary. Both cases use the CloneMeshFVF to accomplish the objective.

```
        case MESH_STANDARD: // Convert to, or continue to use standard mesh type
            if ( m_pPMesh )
            {
                // Attempt to clone the mesh
                hRet = m_pPMesh->CloneMeshFVF( Options, FVF, pD3DDevice, \
                                               (LPD3DXMESH*)&pCloneMesh );
                if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }
           }
            else
            {
                // attempt to clone the mesh
                hRet = m_pMesh->CloneMeshFVF( Options, FVF, pD3DDevice, \
                                              (LPD3DXMESH*)&pCloneMesh );
                if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }
            }
            break;
```

The final case handles cloning to a progressive mesh. If the progressive mesh exists in the source object, then this becomes a straight cloning operation. If not, then we must convert the standard mesh into progressive form, directly storing the result in the output mesh object.

```
        case MESH_PROGRESSIVE: // Convert to, or continue to use progressive mesh type
            if ( m_pPMesh )
            {
                // Attempt to clone the mesh
                hRet = m_pPMesh->ClonePMeshFVF( Options, FVF, pD3DDevice, \
                                                (LPD3DXPMESH*)&pCloneMesh );
                if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }
            }
            else
            {
                // Attempt to clone the mesh
                hRet = D3DXGeneratePMesh( m_pMesh, (DWORD*)m_pAdjacency->GetBufferPointer(),
                                NULL, NULL, 1, D3DXMESHSIMP_FACE, (LPD3DXPMESH*)&pCloneMesh );
                if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }
           }
        break;

    } // End type switch
```

We can now attach the new D3DXMesh to the output CTriMesh object that was passed into the function and release the base mesh interface pointer because we no longer need it.

```
    // Attach this D3DX mesh to the output mesh
    // This automatically adds a reference to the mesh passed in.
    pMeshOut->Attach( pCloneMesh );

    // We can now release our copy of the cloned mesh
    pCloneMesh->Release();
```

Our final task is to test whether the source CTriMesh has any data in its attribute data array. If it does then we are cloning from a managed CTriMesh and we should copy over the attribute data to the output mesh so that it will have the required textures and materials for each of its subsets.

```
    // Copy over attributes if there is anything here
    if ( m_pAttribData )
    {
        // Add the correct number of attributes
        if ( pMeshOut->AddAttributeData( m_nAttribCount ) < 0 ) return E_OUTOFMEMORY;

        // Copy over attribute data
        MESH_ATTRIB_DATA * pAttributes = pMeshOut->GetAttributeData();
        for ( ULONG i = 0; i < m_nAttribCount; ++i )
        {
            MESH_ATTRIB_DATA * pAttrib = &pAttributes[i];
            // Store details
            pAttrib->Material = m_pAttribData[i].Material;
            pAttrib->Texture  = m_pAttribData[i].Texture;
            pAttrib->Effect   = m_pAttribData[i].Effect;

            // Add references so that objects aren't released when either of these
            // meshes are released, or vice versa.
            if ( pAttrib->Texture ) pAttrib->Texture->AddRef();
            if ( pAttrib->Effect  ) pAttrib->Effect->AddRef();

        } // Next Attribute

    } // End if managed

    // Release our referenced D3D Device
    if (pD3DDevice) pD3DDevice->Release();

    // Success!!
    return S_OK;
}
```

It should be noted that the CTriMesh::Attach function has been altered so that it accepts an ID3DXBaseMesh pointer. It will determine whether it is a standard or progressive mesh using IUnknown::QueryInterface and assign the pointer to the underlying ID3DXMesh or ID3DXPMesh member pointer of the CTriMesh object appropriately.

CTriMesh::Optimize

The optimize function has had some small changes to accommodate our dual-use CTriMesh concept. Recall that this function works like a combination clone/optimize-in-place operation. The output mesh type is once again specified using the MESH_TYPE input parameter. The caller must pass in a pointer to an already instantiated CTriMesh object that will receive the new cloned and optimized D3DX mesh.

The first part of this function is the same as CTriMesh::CloneMeshFVF. It exits if the standard or progressive meshes are not yet created, it generates adjacency information if it does not yet exist, and it retrieves the device from the source mesh if the caller passes NULL for the device parameter.

```
HRESULT CTriMesh::Optimize( ULONG Flags, CTriMesh * pMeshOut, MESH_TYPE MeshType ,
                            LPD3DXBUFFER * ppFaceRemap , LPD3DXBUFFER * ppVertexRemap ,
                            LPDIRECT3DDEVICE9 pD3DDevice )
{
    HRESULT             hRet;
    LPD3DXMESH          pOptimizeMesh = NULL;
    LPD3DXBUFFER        pFaceRemapBuffer = NULL;
    LPD3DXBUFFER        pAdjacency = NULL;
    ULONG              *pData      = NULL;

    // Validate requirements
    if ( (!m_pMesh && !m_pPMesh) || !pMeshOut ) return D3DERR_INVALIDCALL;

    // Generate adjacency if not yet available
    if (!m_pAdjacency)
    {
        hRet = GenerateAdjacency();
        if ( FAILED(hRet) ) return hRet;

    } // End if no adjacency

    // If no new device was passed...
    if ( !pD3DDevice )
    {
        // we'll use the same device as this mesh
        // This automatically calls 'AddRef' for the device
        pD3DDevice = GetDevice( );

    }
    else
    {
        // Otherwise we'll add a reference here so that we can
        // release later for both cases without doing damage :)
        pD3DDevice->AddRef();

    } // End if new device
```

Next we allocate an ID3DXBuffer which we can pass into the Optimize function and it will be filled with the face adjacency information for the new optimized mesh. Also, if the ppFaceRemap parameter is not NULL, then the caller would like to know how the faces were re-mapped from their original positions. In that case, we need to allocate an ID3DXBuffer object to contain this information.

```
    hRet = D3DXCreateBuffer( (3 * GetNumFaces()) * sizeof(ULONG), &pAdjacency );
    if ( FAILED(hRet) ) { pD3DDevice->Release(); return hRet; }

    // Allocate the output face remap if requested
    if ( ppFaceRemap )
    {
        // Allocate new face remap output buffer
        hRet = D3DXCreateBuffer( GetNumFaces() * sizeof(ULONG), ppFaceRemap );
        if ( FAILED(hRet) ) { pD3DDevice->Release(); pAdjacency->Release(); return hRet; }
        pData = (ULONG*)(*ppFaceRemap)->GetBufferPointer();
    } // End if allocate face remap data
```

We now optimize the CTriMesh into a new object. Priority is once again given to the progressive mesh and we fall back to the standard mesh if the progressive mesh has not been created. The mesh that is output from the Optimize call will match the type of the mesh that called the function.

```
    if ( m_pPMesh )
    {
        // Attempt to optimize the progressive mesh
       m_pPMesh->Optimize( Flags, (ULONG*)pAdjacency->GetBufferPointer(), pData,
                           ppVertexRemap, &pOptimizeMesh );
    }
    else
    {
        // Attempt to optimize the standard mesh
     m_pMesh->Optimize( Flags, (ULONG*)m_pAdjacency->GetBufferPointer(),
                       (ULONG*)pAdjacency->GetBufferPointer(),  pData, ppVertexRemap,
                        &pOptimizeMesh );
    }
```

We attach the new mesh to the output CTriMesh object passed into the function.

```
    // Attach this D3DX mesh to the output mesh
    // This automatically adds a reference to the mesh passed in.
    pMeshOut->Attach( pOptimizeMesh, pAdjacency );
```

Our next task is to convert the new mesh (if necessary) into the format requested by the caller.

```
    switch ( MeshType )
    {
        case MESH_DEFAULT:  // Continue to use the same type as 'this'

            // Already a standard mesh, does it need converting ?
            if ( m_pPMesh ) pMeshOut->GeneratePMesh(NULL, NULL, 0, D3DXMESHSIMP_FACE, true);
            break;

        case MESH_PROGRESSIVE: // Convert to, or continue to use progressive mesh type

            // Already a standard mesh, convert it
            pMeshOut->GeneratePMesh( NULL, NULL, 0, D3DXMESHSIMP_FACE, true );
            break;

    } // End type switch
```

Now that the new mesh is attached to the output mesh, we can release the temporary pOptimizeMesh interface and the temporary adjacency buffer. Our final task is to copy the attribute array (if it exists) to accommodate managed meshes.

```
    // We can now release our copy of the optimized mesh and the adjacency buffer
    pOptimizeMesh->Release();
    pAdjacency->Release();

    // Copy over attributes if there is anything here
    if ( m_pAttribData )
    {
        // Add the correct number of attributes
        if ( pMeshOut->AddAttributeData( m_nAttribCount ) < 0 ) return E_OUTOFMEMORY;

        // Copy over attribute data
        MESH_ATTRIB_DATA * pAttributes = pMeshOut->GetAttributeData();
        for ( ULONG i = 0; i < m_nAttribCount; ++i )
        {
            MESH_ATTRIB_DATA * pAttrib = &pAttributes[i];
            // Store details
            pAttrib->Material = m_pAttribData[i].Material;
            pAttrib->Texture  = m_pAttribData[i].Texture;
            pAttrib->Effect   = m_pAttribData[i].Effect;

            // Add references so that objects aren't released when either of these
            // meshes are released, or vice versa.
            if ( pAttrib->Texture ) pAttrib->Texture->AddRef();
            if ( pAttrib->Effect  ) pAttrib->Effect->AddRef();

        } // Next Attribute

    } // End if managed

    // Release our referenced D3D Device
    if (pD3DDevice) pD3DDevice->Release();

    // Success!!
    return S_OK;
}
```

CTriMesh::OptimizeInPlace

OptimizeInPlace has also been slightly modified to accommodate our new progressive mesh functionality, although to a lesser degree than the Optimize call. The only different between this version of the function and the version from Lab Project 8.1 is that it tests to see if the object is in progressive mesh or standard mesh mode. In standard mesh mode, this function simply wraps a call to the ID3DXMesh::OptimizeInPlace function for its internal ID3DXMesh object. Since this function does not exist in the ID3DXPMesh interface, we use the less flexible D3DX optimization function called OptimizeBaseLOD to optimize the underlying progressive mesh if the CTriMesh object is in progressive mesh mode.

```
HRESULT CTriMesh::OptimizeInPlace( DWORD Flags, LPD3DXBUFFER * ppFaceRemap ,
                                   LPD3DXBUFFER * ppVertexRemap  )
{
    HRESULT      hRet;
    LPD3DXBUFFER pFaceRemapBuffer = NULL;
    ULONG       *pData = NULL;

    // Validate Requirements
    if ( (!m_pMesh && !m_pPMesh) ) return D3DERR_INVALIDCALL;

    // Generate adjacency if none yet provided
    if (!m_pAdjacency)
    {
        hRet = GenerateAdjacency();
        if ( FAILED(hRet) ) return hRet;
    }

    // Allocate the output face remap if requested
    if ( ppFaceRemap )
    {
        // Allocate new face remap output buffer
        hRet = D3DXCreateBuffer( GetNumFaces() * sizeof(ULONG), ppFaceRemap );
        if ( FAILED(hRet) ) { return hRet; }
        pData = (ULONG*)(*ppFaceRemap)->GetBufferPointer();
    }

    // Optimize the data
    if ( m_pPMesh )
        hRet = m_pPMesh->OptimizeBaseLOD( Flags, pData );
    else
        hRet = m_pMesh->OptimizeInplace( Flags, (DWORD*)m_pAdjacency->GetBufferPointer(),
                                        (DWORD*)m_pAdjacency->GetBufferPointer(),
                                         pData, ppVertexRemap );
        if ( FAILED( hRet ) ) return hRet;

    // Success!!
    return S_OK;
}
```

CTriMesh::SetNumFaces/SetNumVertices

These functions simply wrap the ID3DXPMesh calls of the same names and allow us to dynamically alter the current LOD of the underlying progressive mesh. The function does nothing if the CTriMesh object has not had its underlying ID3DXPMesh generated.

```
HRESULT CTriMesh::SetNumFaces( ULONG FaceCount )
{
    HRESULT hRet = D3DERR_INVALIDCALL;

    // Set number of faces (this is a no-op if there is no pmesh)
    if ( m_pPMesh ) hRet = m_pPMesh->SetNumFaces( FaceCount );

    // Success??
    return hRet;
}
```

```
HRESULT CTriMesh::SetNumVertices( ULONG VertexCount )
{
    HRESULT hRet = D3DERR_INVALIDCALL;

    // Set number of vertices (this is a no-op if there is no pmesh)
    if ( m_pPMesh ) hRet = m_pPMesh->SetNumVertices( VertexCount );

    // Success??
    return hRet;
}
```

CTriMesh::TrimByFaces

TrimByFaces is a dual-mode function that works differently depending on whether the CTriMesh is in progressive or standard mesh mode. In progressive mode, this call simply wraps the call to ID3DXPMesh::TrimByFaces. This will set new upper and lower boundaries for the progressive mesh. The caller specifies the new maximum number of faces, which must be less than or equal to the current maximum number of faces, and the new minimum number of faces which must be greater than or equal to the current minimum number of faces. This function allows us to trim the dynamic range of the progressive mesh freeing collapse structures from memory allowing the mesh to consume less memory.

If this function is called and the CTriMesh is in standard mode (no progressive mesh exists) then the function is interpreted as a request to simplify the underlying ID3DXMesh standard mesh down to a new level of detail. When this is the case, we call D3DXSimplifyMesh to reduce the input mesh (the current standard mesh) to the requested LOD. Since the output of this call is a new mesh, the old standard mesh is released and the new simplied ID3DXMesh is assigned as the new standard mesh for the CTriMesh.

```
HRESULT CTriMesh::TrimByFaces( ULONG NewFacesMin, ULONG NewFacesMax )
{
    HRESULT hRet = D3DERR_INVALIDCALL;

    // Trim by faces
    if ( m_pPMesh )
    {
        // Drop through to P mesh trimming
        hRet = m_pPMesh->TrimByFaces( NewFacesMin, NewFacesMax, NULL, NULL );
    }
    else if ( m_pMesh )
    {
        LPD3DXMESH pMeshOut = NULL;

        // Generate adjacency if not yet available
        if (!m_pAdjacency)
        {
             GenerateAdjacency();
        }

        // Since there is no PMesh, we can only comply by simplifying
        D3DXSimplifyMesh(m_pMesh, (DWORD*)m_pAdjacency->GetBufferPointer(), NULL, NULL,
                         NewFacesMax, D3DXMESHSIMP_FACE, &pMeshOut);
```

```
        // Release old mesh and store the new mesh
        m_pMesh->Release();
        m_pMesh = pMeshOut;

        // Adjacency will be out of date, update it
       GenerateAdjacency();
   }

    // Success??
    return hRet;
}
```

> **Note:** There is also a CTriMesh::TrimByVertices function which is exactly the same code as above but trims LOD based on vertices instead of faces. Check the source code for this project for a listing of this function.

CTriMesh::GetNumFaces

All of the GetXX functions of the CTriMesh class have been slightly modified to accommodate our dual-use mesh class. We will examine the GetNumFaces call in this example, but you should be aware that all of these functions behave in exactly the same way.

In progressive mesh mode, GetNumFaces returns the number of faces being used by the progressive mesh at its current LOD. In standard mesh mode, there are two possible cases. The first is when the ID3DXMesh has been created. In this case GetNumFaces will return the number of faces in the underlying ID3DXMesh. The second case handles the mesh as it is undergoing manual construction, before it has been converted into a D3DX mesh type. In this case we return the number of faces in the temporary index array.

```
ULONG CTriMesh::GetNumFaces( ) const
{
    // Validation!!
    if ( m_pPMesh )       return m_pPMesh->GetNumFaces();
    else if ( m_pMesh )   return m_pMesh->GetNumFaces();
    else                  return m_nFaceCount;
}
```

CTriMesh::DrawSubset

The DrawSubset function has hardly changed at all. ID3DXMesh and ID3DXPMesh both support the DrawSubset function, so this function only needs to determine whether the progressive mesh has been generated or if we should render using the standard mesh. We use an ID3DXBaseMesh pointer to render the subset to handle both cases.

```
void CTriMesh::DrawSubset( ULONG AttributeID )
{
    LPDIRECT3DDEVICE9 pD3DDevice = NULL;
```

```
    LPD3DXBASEMESH    pMesh      = m_pMesh;

    // Retrieve mesh pointer
    if ( !pMesh ) pMesh = m_pPMesh;
    if ( !pMesh ) return;

    // Set the attribute data if it exisits (which means it is a managed mesh)
    if ( m_pAttribData && AttributeID < m_nAttribCount )
    {
        pD3DDevice = GetDevice( );
        pD3DDevice->SetMaterial( &m_pAttribData[AttributeID].Material );
        pD3DDevice->SetTexture( 0, m_pAttribData[AttributeID].Texture );
        pD3DDevice->Release();
    }

    // simply render the subset(s)
    pMesh->DrawSubset( AttributeID );
 }
```

Additional Code Changes

The rest of the changes to our scene viewer can be found in the CScene class. Most changes are relatively minor and you will find most of the code from Lab Project 8.1 intact.

The first change is the addition of one line of code to the ProcessMeshes and ProcessReference functions. Once these functions have created the CTriMesh and generated the underlying ID3DXMesh, we call the new function CTriMesh::GeneratePMesh to generate the underlying progressive mesh (placing it into progressive mesh mode) before adding it to the scene mesh array. The following snippet from ProcessReference illustrates the idea:

```
…
CTriMesh * pNewMesh = new CTriMesh;

// Load in the externally referenced X File
pNewMesh->RegisterCallback( CTriMesh::CALLBACK_ATTRIBUTEID, CollectAttributeID, this );
pNewMesh->LoadMeshFromX( Buffer, D3DXMESH_MANAGED, m_pD3DDevice );

// Attempt to optimize the standard mesh ready fro progressive mesh generation
pNewMesh->WeldVertices( 0 );
pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );

// Convert this mesh to a progressive mesh
pNewMesh->GeneratePMesh( NULL, NULL, 50, D3DXMESHSIMP_FACE );

// Optimize the progressive form of the mesh
pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );

// Store this new mesh
if ( AddMesh( ) < 0 ) { delete pNewMesh; return false; }
m_pMesh[ m_nMeshCount - 1 ] = pNewMesh;
…
```

In this demo, all of the meshes in our scene mesh array will be progressive meshes with an arbitrarily set minimum LOD of 50 faces. Note that the skybox will not be a progressive mesh.

CScene::CalculateLODRatio

We have added a few new functions to our CScene class to set the current LOD for our meshes prior to rendering. The LOD selected will be based on the distance from the mesh to the camera. As meshes get further away from the camera, they will use fewer faces and vice versa. This is easiest to see if you switch the application to wireframe render mode.

CalculateLODRatio manages this process. The function returns a value between 0.0 and 1.0 based on camera distance. This value will be used to select a new LOD for the number of mesh faces in the CScene::Render function. The [0.0, 1.0] range serves as a percentage between the minimum and maximum number of faces for the mesh.

```
float CScene::CalculateLODRatio( const CCamera & Camera, const D3DXMATRIX & mtxObject ) const
{
    // Retrieve object position.
    D3DXVECTOR3 vecObject = D3DXVECTOR3( mtxObject._41, mtxObject._42, mtxObject._43 );

    // Calculate rough distance to object
    float Distance = D3DXVec3Length( &(vecObject - Camera.GetPosition()) );

    // Calculate the LOD Ratio, from 0.0 to 1.0 within the first 60% of the frustum.
    float Ratio = 1.0f - (Distance / (Camera.GetFarClip() * 0.6f));
    if ( Ratio < 0.0f ) Ratio = 0.0f;

    // We have our value
    return Ratio;
}
```

We pass in the camera and the world matrix for the object we are about to render. The world space position is extracted from the fourth row of the matrix and the distance to the camera is calculated using the D3DXVec3Length function. We then divide the distance by the distance to the camera far plane and multiply by 0.6. This means that any mesh that is over 60% of the distance from the camera, between the near and far planes, will always have a ratio of 0.0. In this case the mesh will be set to its lowest level of detail.

CScene::Render

The CScene::Render function has not been modified much since Lab Project 8.1 so we will examine only the changes that were made. Three new lines were added to the call (highlighted in the following listing) to manage the LOD transition for each mesh.

For each mesh, we calculate its LOD ratio by passing its object world matrix to CalculateLODRatio. The return value is used to calculate the desired face count for the progressive mesh. The ratio interpolates between the minimum and maximum number of faces for the mesh. We then set the mesh LOD with a call to the CTriMesh::SetNumFaces.

```
    for ( i = 0; i < m_nObjectCount; ++i)
    {
        CTriMesh * pMesh = m_pObject[i]->m_pMesh;

        // Calculate LOD ratio
        float Ratio = CalculateLODRatio( Camera, m_pObject[i]->m_mtxWorld );

        // Calculate the number of faces (simple linear interpolation)
        ULONG FaceCount = pMesh->GetMinFaces() +
                          (ULONG)((float)(pMesh->GetMaxFaces()–pMesh->GetMinFaces()) * Ratio);

        // Set the LOD's number of faces
        pMesh->SetNumFaces( FaceCount );

        // Setup the per-mesh / object details
        m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i]->m_mtxWorld );
        m_pD3DDevice->SetFVF( pMesh->GetFVF() );

        // Loop through each scene owned attribute
        for ( j = 0; j < m_nAttribCount; j++ )
        {
            // Retrieve indices
            MaterialIndex = m_pAttribCombo[j].MaterialIndex;
            TextureIndex  = m_pAttribCombo[j].TextureIndex;

            // Set the states
            if ( MaterialIndex >= 0 )
                m_pD3DDevice->SetMaterial( &m_pMaterialList[ MaterialIndex ] );
            else
                m_pD3DDevice->SetMaterial( &m_DefaultMaterial );

            if ( TextureIndex >= 0 )
                m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ]->Texture );
            else
                m_pD3DDevice->SetTexture( 0, NULL );

            // Render all faces with this attribute ID
            pMesh->DrawSubset( j );

        } // Next Attribute

        // Increase tri-rendering count
        m_nTrisRendered += pMesh->GetNumFaces();

    } // Next Object
```

Notice that while the progressive mesh provided by D3DX is a view *independent* progressive mesh, we calculate the LOD based on view dependant information. This is a common strategy, but not the only way to do it. You might instead enable a menu option for reducing/increasing LOD. Perhaps certain meshes are considered more important than others and the lower priority meshes can be simplified. The

bottom line is that how you choose to modify LOD is up to you. Here we have implemented a distance based approach simply to show how the CTriMesh in progressive mesh mode can be used by an application.

We now have a CTriMesh class that is much more than a simple wrapper. Indeed, it is more like a complete mesh toolkit handling resource management, standard and progressive meshing techniques, optimization and cloning from standard to progressive meshes and vice versa and finally, automated or scene handled rendering strategies are exposed to the application. This functionality will serve us well in future applications as we move forward in the course.

Chapter Nine

Frame Hierarchies

Introduction

In Chapter Eight, while examining the process of loading and rendering mesh data stored in X files, we learned that the D3DXLoadMeshFromX function provided a useful and yet restrictive service: regardless of how many meshes were contained in the X file, a single mesh was always output. Multiple meshes, were they to exist in the file, were simply transformed and collapsed into a single output mesh. While this outcome may be acceptable under certain conditions, a more flexible X file loading system that allows us to optionally maintain the individual mesh entities would certainly be ideal. This is because there are circumstances that frequently occur in game development where the preservation of the distinct meshes is preferred. We will examine a number of these cases as we move forward in this chapter.

For example, consider the case where an X file is used to store the representation of an automobile. It is not uncommon for an automobile model to be constructed using several meshes (ex. chassis, doors, trunk, hood, and tires), each requiring independent animation. The game designer may want the wheels to rotate as the car accelerates or the doors to swing open and closed when the player enters or exits the vehicle. It would be difficult indeed to apply such localized animations to the various sections of the automobile if a single mesh representation was used. Earlier lessons have taught us that when we apply a transform to a mesh, it is applied to all of its vertices. Therefore, if we wish to represent a model that has independently moving parts, it should be represented as a collection of unique meshes that, taken together, represent the overall model. With a multiple mesh representation, each mesh is rendered separately. This affords us the freedom to alter the transformation matrix for a given mesh before it is rendered, applying animation to position, orientation, or even both simultaneously.

The X file loading techniques we have discussed thus far would not be able to be used in circumstances where such a representation is stored in the X file. Using the D3DXLoadMeshFromX function, we would effectively abandon the multi-mesh layout of the file, getting back a single mesh that is the sum of all specified submeshes. While the process of collapsing submeshes into a single mesh is handled intelligently by D3DX (the original submeshes are correctly positioned within the overall model representation) we will have lost the ability to animate/transform those original sub-meshes separately from the rest of the model.

When constructing or loading a multi-mesh representation, we do not simply want the various sub-meshes stored such that they are transformed and rendered in total isolation from one another. Rather, it would be expected that the underlying spatial relationship between each of these individual meshes will still be preserved, even when these submeshes experience their own unique transformations. If the vehicle moves, then all of its submeshes must move together, to maintain model integrity. We would not want to move the automobile chassis mesh only to find that the wheels have not moved along with it. However, we do want the freedom to rotate the wheels without rotating the chassis.

Designing such an automobile representation in any of the popular 3D modeling packages is quite common, and we are generally going to want to use a multi-mesh representation in our application as well. One important question is, when this automobile is exported to the X file format, how are these meshes arranged in the file such that they can be imported and assembled into a multi-mesh vehicle model that behaves as expected? The quick answer is that each mesh in the file will have a

transformation matrix assigned to it that describes how it is positioned relative to something else in the scene. This 'something else' might be another mesh used in the model, or some other point in space (perhaps even the scene origin itself). The result is a spatial hierarchy of meshes, arranged in a tree-like fashion. There is a root node, and then a series of child nodes positioned relative to the root. These children may have child nodes positioned relative to themselves, and so on.

Using our automobile example, the chassis mesh might be stored in the root node of the tree and the four wheel meshes might be stored as child nodes of the root. In each child node, a wheel submesh would also be accompanied by a 4x4 transformation matrix which describes the position and orientation of the wheel mesh. This description is *not* an absolute (i.e. world space) transformation as we might expect, but is in fact a transformation *relative* to the parent node; in this case, the chassis. In this simple example, the transformation matrix for each wheel mesh would describe the position and orientation of the wheel relative to the location and orientation of the chassis mesh (the parent node). Therefore, regardless of how we animate the chassis (by applying transformations to the matrix in the root node), the world matrix used to render each submesh can always be calculated such that object integrity is maintained.

In this example, the root node matrix is used to set the world space transformation for the chassis mesh. This is essentially the world matrix for the entire automobile representation. It is this matrix that would be used to transform the chassis mesh into world space prior to rendering it. The world space matrices used to render each submesh are *not* directly contained in the representation (or in the X file for that matter) and must be calculated each time the root node matrix is updated by the application. We accomplish this by combining a parent node matrix (the chassis/root matrix in this example) with the child transformation matrix (a wheel matrix) prior to rendering. As each child wheel matrix describes the position of the wheel relative to the chassis, and the root node matrix describes the world space position of the chassis itself, multiplying these two matrices produces the world space transformation matrix for the wheel. By generating the world matrices for each submesh at runtime, just prior to rendering, we are assured that whenever we apply a transformation to the root node matrix, the world matrix generated for each child mesh will correctly transform the child along with the parent. This ensures that the spatial relationship between the children (the wheel meshes) and the parent (the chassis mesh) is maintained and that changes to the root node matrix are propagated through the entire tree. The result is that we can move the entire model, with all of its submeshes, by applying transformations to the root node matrix and then calculating the new world space matrix for each child mesh that results from these changes to the root. Note as well that we also have the ability to apply transformations to the relative matrix of a child object. In this manner, local rotations could be applied to our wheel meshes without being applied to the chassis.

Graphics programmers commonly refer to the data arrangement that describes these relative relationships as a *spatial hierarchy* or a *frame hierarchy*. Understanding how to load, animate, and render a multi-mesh frame hierarchy is vital if we are to progress past the simple mesh representations we have covered thus far. To understand the relative arrangement of a frame hierarchy, we must first examine the concept of *frame of reference*. This will be the subject of the next section.

9.1 Frame Hierarchies

Physicists and mathematicians use the concept of a *frame of reference* (a.k.a. *reference frame*) to describe an arbitrary point in space with an arbitrary orientation. We use frames of reference all the time in our everyday lives to achieve all manner of tasks and as such, we are often not aware that we are using them at all. Without a frame of reference it would be impossible to evaluate a grid coordinate on a map, give somebody directions to your house, or even specify a position in 2D or 3D space.

A simple example of why we need to use a frame of reference is apparent when we consider giving somebody travel directions. Imagine that Person A lives in San Francisco and that Person B resides in Miami. If A asks B the general direction he should travel to reach Mexico City, B might incorrectly assume that they both live in Miami and offer information describing Mexico City as being positioned due west. If A also assumes that B is living in San Francisco and is offering his directions based from that location, A would head off in a westerly direction and find himself in the middle of the Pacific Ocean. Of course, Person A could have stayed dry if B had revealed that his directions were using Miami as a frame of reference. The correct directions should have been given as "From Miami, head west". In this case, Person A could have traveled to Miami first and then followed the directions provided, or he could have extrapolated his own directions from the Miami-based directions. This would have informed him that he needed to head due south and then east.

Another common example of frame of reference is one which demonstrates how the perceived velocity of a moving object is changed when classified using different frames of reference. This example will also have some relevance to 3D animation, as we will learn later. To demonstrate, imagine Person A (who we shall refer to as Speedy Steve) riding a motorcycle up the street in a northerly direction at a speed of 150 kph. Just as he passes by Person B (who we shall call Dastardly Dennis), Dennis walks out into the center of the street and positions himself directly behind Speedy Steve. He then pulls back on his super-slingshot as hard as he can, and releases a stone in the direction Speedy Steve is traveling. The stone leaves the slingshot and travels at a constant 155 kph. Dennis is a rather nasty fellow whose intention is to strike Speedy Steve in the back of the head so hard that he is knocked from his motorcycle. However, what Dennis had cruelly planned does not come to fruition. Instead, the stone gently taps Speedy Steve on the back of the helmet, so softly that he hardly notices it. He carries on northbound and vanishes out of sight. Dennis is left scratching his head and wondering why his dastardly plan failed.

The reason Dennis' plot did not work becomes clear when we examine the nature of the relative velocities. Although the stone that Dennis fired was traveling at a frightening 155 kph in Dennis' frame of reference, Speedy Steve is also traveling at 150 kph himself in the same direction. So from Steve's perspective, the stone is only gaining on him at a relative velocity of (155 – 150) 5 kph. That it, using Speedy Steve as a frame of reference, the velocity of the stone is only 5 kph. Therefore it strikes him with very little force. Of course, if Speedy Steve was stationary like Dennis, then the stone would have hit him at 155 km per hour and probably done some serious damage. For maximum dastardly effect, Dennis should have preempted Steve's arrival and been laying in wait. If he would have released the stone at the same speed from in front of Steve, and in a southerly direction, it would have hit Steve head-on with devastating force. Using Steve as a frame of reference in this example, what would be the velocity of the stone at the point of impact? The answer is (155 + 150) 305 kph. This is because both

Steve and the stone are traveling in exact opposite directions and therefore their combined velocity is used to calculate their impact velocity. As Steve hurtles towards the stone, it in turn is hurtling towards him. So from Steve's point of view, that stone is getting closer at a very high rate of speed indeed. The main point is that the position, orientation and velocity of something can actually be very different when observed using different frames of reference.

As it turns out, we have consistently used frames of reference throughout this course series when dealing with mesh data and transformations, although their application is so obvious it might be overlooked. The 3D vector <10, 10, 10> would mean nothing to us, and could not even be evaluated, if we did not know that these numbers describe a position in space relative to the origin of some coordinate system. Indeed, a coordinate system provides a means for describing locations within a frame of reference. When plotting points in any coordinate system (world space, view space, projection space, screen space, etc.) we are using a frame of reference with the origin of the coordinate system serving as the origin of the frame. We know that this is usually classified as position (0, 0, 0).

Throughout Graphics Programming Module I we discussed mathematical 'spaces' and coordinate systems in some detail. So we see now that in effect, we have been discussing frames of reference all along. Each of the 'spaces' we discussed (model space, world space, view space, etc.) had an origin and orientation which described a frame of reference. All positions in that space are defined relative to this origin and axis orientation. In model space, all vertex positions are defined relative to the fixed position (0,0,0) and the coordinate axes are the standard orthogonal XYZ axes. In world space, those same vertices are then defined relative to another fixed point in space (also assigned the coordinates (0,0,0)) with its own orthogonal set of XYZ axes. This point is considered the origin of our world space system and is therefore the frame of reference for all vectors plotted in world space. View space is yet another frame of reference that exists at the camera position (in which (0,0,0) is also used as the frame of reference origin for all points described within that space). It too has its own set of XYZ orthogonal axes which, with respect to the world's standard axes, describe the orientation of the camera.

It should now be clear then that transformations provide us with a means for modifying values within a frame of reference. If we consider the three transformations we have studied in most detail (scaling, translation, and rotation) we understand that they can be used to modify the position of a point in order to generate a new position within the same coordinate space (frame of reference). Transformations also provide us the ability to transform (or perhaps more appropriately, "reclassify") values from one frame of reference to another. We have seen how the world space transformation transitions vectors relative to the model space frame of reference into vectors relative to the world space frame of reference. Many people actually find this a more intuitive way of visualizing transformations from one space to another. We are simply taking a point in one frame of reference and reclassifying it in another.

So then a frame of reference is essentially a *local space*: it has an origin from which emerges a set of axes that define how the space is oriented and thus how things in that local space are measured with respect to that origin. Through the use of transformations we can create a situation in which we redefine points in one frame of reference to be classified with respect to another frame of reference. That is how our models maintain a presence in our game world. Each model usually has its own frame of reference (model space, in which its vertices are initially defined), but through our transformation pipeline, we can also describe how those same vertices are situated in the world frame of reference, the camera frame of reference, and so on.

The reason why an understanding of frames of reference is so important can be demonstrated when we examine our automobile example. We will use a simple example of five separate meshes (four wheels and a car body) for our automobile. These meshes will be arranged in the X file in a hierarchical format such that the positions of the four wheels are defined relative to the position of the car body. In other words, the car body is stored at the root of the automobile hierarchy and the four wheels will be children of this node.

In Fig 9.1 we see the automobile hierarchy. In the diagram, we see a **frame** structure which will represent a node in the tree. A single frame contains a 4x4 transformation matrix defining a location and orientation as just described and stores pointers to *N* child frames. Each frame (short for 'frame of reference') can also store one or more pointers to data items -- car part meshes in this case. In this example, each frame contains only a single child mesh, but this need not always be the case.

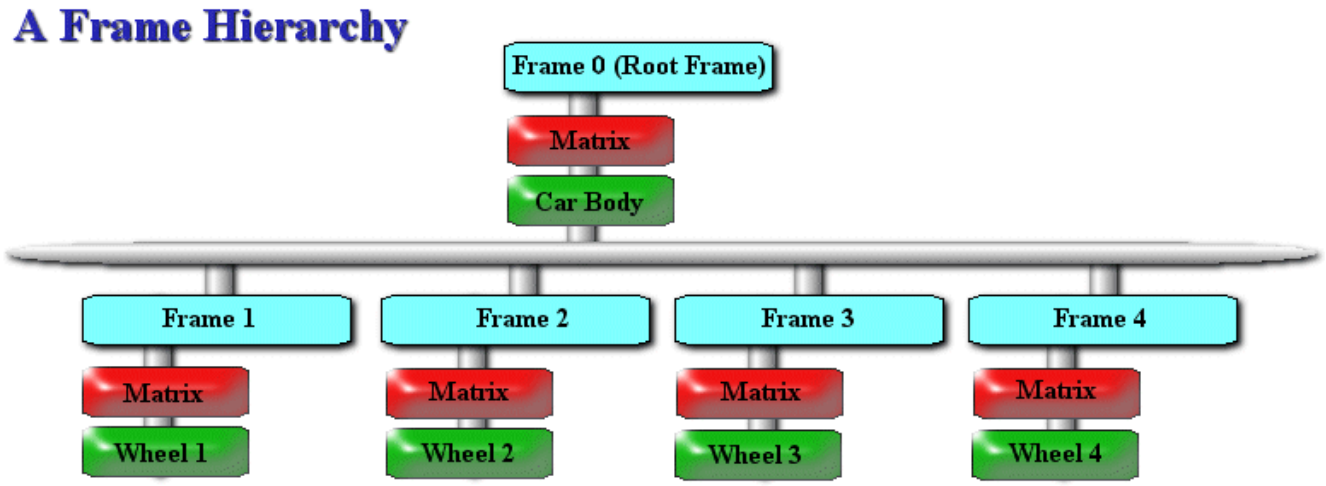


Figure 9.1

Each 4x4 frame matrix consists of two components. It has a translation component in its bottom row which describes the origin of the frame of reference. This origin is defined *relative to* its parent (i.e. it tells us the location of the origin of the child frame within the parent's frame of reference). The rest of the matrix describes the parent-relative orientation of the frame itself (i.e. how much rotation off the parent axes the child axes happen to be). The combination of parent-child relationships defined by the transformations stored in each node matrix creates a spatial hierarchy.
While the root node has no direct parent in the hierarchy, we can still think of its matrix as a relative matrix. In this particular case, the root node 'parent' is the world system origin, and the root frame matrix stores a position/orientation relative to that system. As such, the root frame transformation matrix will serve as a world matrix for the entire hierarchy, enabling us to position and orient the entire set of hierarchy data in our world (assuming the root node does not get attached to some other parent of course – for example, an automobile carrier that is transporting cars to the showroom). This is possible because the children of the root frame are defined in its local space (i.e. reference frame), so by moving the root frame in the world, we are moving its frame of reference and thus any items that are defined within it.

So in our simple case, applying a translation to the root frame would position that frame somewhere in the game world. Certainly we would expect that the children must also experience the same transformation so that they follow their parent to the new location (the spatial dispositions of the child

objects in the hierarchy must be preserved in the process of moving their parent). Keep in mind that while the root node world transformation must be propagated down to each child so that they move as well, the relative relationships between parents and children must remain intact since they exist within that given reference frame. Thus the world transformation and the parent-relative transformations at each node must be combined in some fashion. We will examine the appropriate matrix combination strategy shortly.

The root frame in this particular example stores a single mesh (the car body) and its transformation matrix describes the world orientation and position of the frame. Thus to transform the car body mesh vertices into world space, we would just multiply the vertices by the frame 0 matrix directly. In that sense, we can think of the mesh stored in the root frame as having a local space defined by that root frame (i.e. the mesh vertices are defined as if the root frame origin were the model space origin for the mesh itself). We will see shortly that in fact, this holds true for all of our frames and that we are essentially building a tree of local space coordinate systems all nested within each other.

We will want to store each wheel in a separate frame so that rotation can be applied to the individual wheels when the car accelerates. This would be a local rotation for the wheel, likely around its local X or Z axis depending on how the wheel model was oriented in the modeling program. Thus matrix multiplication order is very important here since the frame matrix stores a translation that already positions the wheel away from the center of the car body. Of course, we would not want that translation to happen first, since that would just rotate the wheel around the parent. Thus the local wheel rotation would have to happen before the wheel was translated into its world space position. In other words, we want to start with the wheel at the origin, rotate it around its local X or Z axis so that it spins, and then push that wheel out into its final position relative to its parent. So the local rotation matrix would have to come first in the multiplication order, followed by the parent-relative frame matrix, and not the other way around.

For simplicity, let us forget about the mesh vertices and work only with mesh center points. In a sense we can even think of each mesh as a single vertex in a model called ‘scene’. Let us further assume that we would like the car to be moved to world location (100,100,100) and we define a 4x4 translation matrix **W** to accommodate this.

If we assume that our car body is centered at the origin of the model and thus requires no additional transformations relative to the center of the model, its parent-relative (i.e. frame) transformation matrix would be an identity matrix (which we can call **F0** for Frame 0). As mentioned above, this will actually be our root frame matrix in the case of the car body since this mesh stored in the root frame.

Let us call the final world space transformation matrix for the car body **B**. This will be the matrix that we use to transform the car body mesh into world space.

B = **F0** * **W**
B = W (since F0 is an identity matrix)

As you can see, all we have done above is multiply the matrix in the root frame with translation matrix **W**. This results in matrix **B**, which is a matrix describing a world space translation of (100,100,100) to move the car body vertices into world space.

Now let us examine one of our wheel frames and say that in this example, each wheel is offset 10 units from the center of the car body. Thus each wheel frame will have a 4x4 translation matrix defining its relationship to its parent. This is the frame matrix for the wheel and we will call it **F1** (for the first wheel).

The question is, how do we find the final *world space* transformation for the wheel (we will call it **T1**)? F1 does not describe the position of the wheel using the world space frame of reference, but that is exactly what we need in order to transform its vertices into world space prior to rendering. Therefore, the wheel frame matrix must be multiplied with something else to generate the final world space transformation matrix for the wheel.

T1 = **F1** * ?

We know that matrix **F1** contains a transformation relative to its parent **F0**, so these will have to be multiplied together so that both meshes are in the same space and share a common frame of reference. However, as we have potentially repositioned **F0** in the world by multiplying it with translation matrix **W**, the final world space position of the wheel must take this into account also so that the wheel and the car body frames are translated by the same amount.

T1 = **F1** * (**F0 * W)**
T1 = F1 * B

As you can see, **F1** describes its position relative to **F0**, so multiplying these matrices together will move the wheel from its own local space into the local space of the entire automobile representation. We can imagine at this point that the wheel has been correctly positioned relative to the car body but that the car itself is still at the origin of world space. Finally, **W** contains the transform matrix that positions the entire model in the world (we used this to position the car body mesh) so this too must be applied to the wheel mesh. Therefore, the car body world matrix is **B=F0*W** and the world matrix of the wheel is **F1*B**.

So if we wanted to rotate the wheel around its own local Z axis, it should be apparent to you where that 4x4 rotation matrix ($\mathbf{Z_{rot}}$) would need to go in the chain:

T1 = $\mathbf{Z_{rot}}$ * **F1** * **F0 * W**

Now in this particular case, the parent of the wheel is the root node of the hierarchy. Thus the equation simplifies to:

T1 = $\mathbf{Z_{rot}}$ * **F1** * **B**

Hopefully you see the pattern here. If you wanted to attach something to the wheel (a hubcap for example), you would add a new frame for it and define it relative to its parent (the wheel) and continue on down the chain.

It is important to keep in mind that any transformation at a node affects all of its children when we combine the matrices in this way to generate world matrices for each frame (and the meshes they may

contain). In that sense, it can be helpful to think of frames as a system of joints. Any part of the scene that needs to be moved or rotated independently will be assigned its own joint. The connections between those joints are essentially offsets of some distance from the parent, represented by the translation row in the frame matrix. How those joints are oriented would be represented in the upper 3x3 portion of the matrix (an orthogonal rotation matrix). Thus at any given node in the tree, its world transformation matrix is defined as the concatenation of any local transforms we wish to apply to that frames matrix, the parent-relative transformation matrix stored in the frame itself, and the parent's current **world** space transformation matrix (which is itself a by-product of the same logic). Therefore, in order to calculate the world space transformation matrix of a given frame in the hierarchy, we must have first calculated the world space transformation matrix for its parent frame, as it is used in the concatenation.

In our automobile example we saw that each wheel has its own frame of reference (Fig 9.1). We now also know that a frame matrix stores a mathematical relationship describing some translational offset from a parent frame and a local orientation for that node which is relative to the parent node's orientation. Practically speaking, we can consider the mesh vertices that are stored at a given frame as children of that frame as well. That is, they utilize the same matrix concatenation scheme to describe how far away they need to be from the frame root node. The only real difference is that the vertices will have their final matrix concatenation done in the pipeline during rendering (just as we saw in Graphics Programming Module I) while child frames have to calculate their new matrix information manually as we descend the hierarchy. In reality though, both are experiencing the same thing -- offsetting and orienting with respect to their parent, which itself was offset with respect to its parent and so on up the chain. Also remember that while the root frame's matrix will often serve as the world matrix for any immediate child meshes of the root frame (again, the parent of the root frame is often going to be the world origin itself) we now know that this is not the case for meshes stored at subsequent child frames -- their world matrices must be generated by stepping down the tree to those frames, concatenating any direct or indirect parent matrices as we go.

Before wrapping up, let us step through our automobile example one more time, using some real values to try to clarify the important points.

If we reconsider Figure 9.1, we can imagine the root frame (we will assume the root frame has no specified parent in the scene and thus defaults to having the world origin as its parent frame) might have a matrix that stores a translation vector of (100,100,100). This then describes the center of the entire automobile as being at location (100,100,100) in world space. We will assume for now that the matrix contains no rotational information and that the car body has been designed such that its front faces down the world Z axis when no rotations are being applied. Now, let us assume that the wheels themselves are placed at offsets of 10 units along both the automobile's X and Z axes, such that the front driver's side wheel for example (wheel 1) is positioned (assuming a steering wheel on the left hand side) at offsets <–10,0,10> from the automobile center point, the front passenger wheel (wheel 2) is positioned at offset <10,0,10> from the automobile center point, the driver side rear wheel (wheel 3) is positioned at offset <–10,0,-10> and finally the passenger side rear wheel is located at offset <10,0,-10> from the automobile center point. Remember that the world space center point of the automobile is the root frame position described by its matrix.

Prior to our previous discussion, we might well have assumed that the information would be stored in the X file such that the matrices of each wheel frame would describe the absolute world space position

of that wheel. That is, it would have been easy to assume that the five matrices in this example had their translation vectors (the 4th row of each matrix) represented as:

Matrix Translation Vectors (This is not correct)

Root Matrix	= **100 , 100 , 100**	(The World Matrix of the Hierarchy)	
Wheel 1 Matrix	= **90 , 100 , 110**	(World Matrix offset by	–10, 0, 10)
Wheel 2 Matrix	= **110 , 100 , 110**	(World Matrix offset by	10, 0, 10)
Wheel 3 Matrix	= **90 , 100 , 90**	(World Matrix offset by	-10, 0, -10)
Wheel 4 Matrix	= **110 , 100 , 90**	(World Matrix offset by	10, 0, -10)

If the wheel frames contained absolute matrices such as these, which described the complete world space transformation for any child objects of the frame (the wheels in this example), then rendering the hierarchy would be very straightforward:

1. set root frame matrix as device World Matrix
2. render any child meshes of the root frame (the car body)
3. for each child frame of the root frame
4. set child frame matrix as device World Matrix (a wheel matrix)
5. render the wheel mesh for that child

While this is certainly easy enough to code (and in fact, we have done this throughout the last course module and in the last chapter), this is no longer the hierarchy we envisioned. In this case, all frames share the same parent since they are all defined relative to the same point in space (the world origin). What was a hierarchy has now devolved into a set of standard world space meshes. The relationship between the car body and the wheels exists in name only, for the children are not affected by changes to their assigned parent. In terms of their behavior, the world is their parent since that is what they are defined relative to.

However, if we were to examine the translation vector of each of the five matrices in a proper automobile X file, we would correctly find that the wheel matrices would actually describe transformations relative to their parent (the root frame in this case). Again, we would notice that the root frame has no parent frame so can be thought of as being a transformation relative to the origin of world space and serves as the base world matrix for the entire hierarchy:

Matrix Translation Vectors (Correctly stored relative matrices for child frames)

Root Matrix	= **100 , 100 , 100**	(The World Matrix of the Hierarchy)
Wheel 1 Matrix	= **-10 , 0 , 10**	(RootMatrix * Wheel1Matrix = 90 , 100 , 110)
Wheel 2 Matrix	= **10 , 0 , 10**	(RootMatrix * Wheel2Matrix = 110 , 100 , 110)
Wheel 3 Matrix	= **-10 , 0 , -10**	(RootMatrix * Wheel3Matrix = 90 , 100 , 90)
Wheel 4 Matrix	= **10 , 0 , -10**	(RootMatrix * Wheel4Matrix = 110 , 100 , 90)

Using the render approach described previously would obviously not work any longer. If we were to simply set wheel matrix 1 as the world matrix before rendering wheel 1 then the wheel would be rendered at world space position (–10,0,10) while the car body sits at position (100,100,100). Also, when we wish to move the automobile about in the world, ideally we only want to translate or rotate the

root frame and let the children experience these updates appropriately. As discussed previously, for this to work we must move through the hierarchy one level at a time, concatenating matrices as we go. So we would start at the root level, set the root frame's matrix and render any immediate child meshes -- in this case the car body. At this point the car body would be rendered at world space position (100,100,100). Next we would iterate through each child frame of the root. For each child frame we create a temporary matrix by multiplying its frame (i.e. parent-relative) matrix by the root frame matrix. In the case of wheel 1 for example, the translation vector of this combined matrix in the above example would now be (90,100,110). This clearly describes a world space position, but it also shows us that this wheel remains correctly offset (-10,0,10) from the current root frame position. We then set this temporary combined matrix as the device world matrix and render that wheel. We repeat this process for each additional wheel.

Since a given hierarchy could be many levels deep, we see right away that this concatenation/rendering approach would be nicely served by a simple recursive function call. When the matrix of any frame in the hierarchy is altered, these changes will automatically be propagated down through all child frames as we traverse the hierarchy and combine the matrices at each frame prior to rendering their immediate child meshes.

If we were to change the root frame matrix in our above example to some new position, then the next time we render the hierarchy and step through the levels of the hierarchy combining matrices, the world space positions of all the wheels would be updated too -- even though we do not physically alter the matrices of these child frames. This last point is important to remember. If our application were to alter the root frame matrix so that it now had a translation vector of <50,0,50>, then the next time the automobile is rendered using the concatenation technique just described we know certain things will be true. For starters, the car body would be rendered with its center point at (50,0,50). This is obvious because for the root frame meshes we simply set the root frame matrix as the world matrix. When we render the wheel 1 mesh, we create a temporary world matrix by combining the root frame matrix with the wheel 1 frame matrix (which still has a translation vector of <–10,0,10> and has not changed) and use this to render the wheel 1 mesh as before. This time, the combined matrix would have a translation vector which places the wheel at position (40,0,60) in the world, still an offset of (-10,0,10) from the root matrix position (50,0,50). So while the wheel matrices have not changed, the temporary matrices we generate to render the frame's meshes have changed. The result is that the wheels will automatically move along with the car body as expected and maintain their proper spatial positioning at all times.

Matrix Translation Vectors (Root Matrix now contains position 50,0,50)

Root Matrix	=	**50, 0, 50**	(The World Matrix of the Hierarchy)
Wheel 1 Matrix	=	**-10, 0, 10**	(RootMatrix * Wheel1Matrix = 40 , 0 , 60)
Wheel 2 Matrix	=	**10, 0, 10**	(RootMatrix * Wheel2Matrix = 60 , 0 , 60)
Wheel 3 Matrix	=	**-10, 0, -10**	(RootMatrix * Wheel3Matrix = 40 , 0 , 40)
Wheel 4 Matrix	=	**10, 0, -10**	(RootMatrix * Wheel4Matrix = 60 , 0 , 40)

The same logic holds true for any rotations or scaling we may apply to any matrix in the hierarchy. If we were to rotate the root frame 45 degrees clockwise about its local Y axis (by creating a Y axis rotation matrix and multiplying it with the root matrix), then when we traverse the hierarchy combining matrices as we go from level 1 to level 2, the temporary matrix created for each wheel will also rotate the wheel by the same amount. The offset of <–10,0,10> for wheel 1 for example is still preserved of course. The

result is that as the car body rotates, the wheels rotate too -- not about their own center points but about the center point of the car body. This is because the frame of reference they are defined in is experiencing its own local rotation, thus dragging its children along with it. This allows the children to remain in their correct position at all times.

Seeing some example code on how to traverse and render a hierarchy should cement all of this into place. This is not our final code of course because we have not yet discussed how to load the hierarchy from an X file or what support structures we may need, but we can imagine that each frame in the hierarchy could be represented using a structure a little like the following:

```
struct CFrame
{
      LPSTR             pFrameName;
      D3DXMATRIX        Matrix;
      CSomeMeshClass *  pMesh;
      CFrame *          pChildFrame;
      CFrame *          pSiblingFrame;
};
```

Keep in mind how the frames are connected together and that each frame contains a pointer to a mesh which is assigned to that frame. Everything stored in this frame, be they vertices or other frames are all children of the frame. That is, they are live in its local space and are defined relative to its origin and are affected by its orientation. In our example, if this was the root frame, then this mesh pointer would point to the model for the car body.

The pChildFrame pointer points to a linked list of child frames, but it is important to note that the linked list is not organized in the way you might expect. Rather, you should think of the pSiblingFrame member as being the pNext member of a traditional linked list and then visualize each tier in the hierarchy as a separate linked list. So the linked list of one level will be attached to its parent level using the pChildFrame pointer of the parent frame. Using our hierarchy as an example, the root frame would have its pSiblingFrame pointer set to NULL because the root frame is not actually part of any linked list of frames at that level in the hierarchy (there is only one frame at the root level – Level 0). The root frame's pChildFrame pointer would point to Frame1 (the first wheel), linking the root level to Level 1. Frame 1 would have its pChildFrame point set to NULL (as will frames 2, 3, and 4) because it has no additional child frames. However, Frame1 will connect to Frame2 via its pSiblingFrame pointer. Frame2 will have its pSiblingFrame pointer pointing to Frame3. Finally, Frame3 will have its pSiblingFrame pointer pointing to Frame4. Frame4 will have a NULL pSiblingFrame pointer since it is the end of the list at that level in the hierarchy.

In summary, the pChildFrame pointer for a given frame structure points to the first child frame in a linked list of child frames and represents another level in the tree. Each child frame in the linked list would be attached by their pSiblingFrame pointer. From this we can understand that all frames linked by their pSiblingFrame pointers belong to the same level in the hierarchy and share the same parent. Their transformation matrices will define their positions and orientations relative to the same frame of reference, which would be the world space position described by the parent frame. In our automobile example, the hierarchy has two levels: the first level contains only the root frame, the second level contains the four wheel frames. The wheels are all considered siblings because they share the same parent.

In order to render this hierarchy, we could use a recursive function like the following:

```
void DrawHierarchy(CFrame *pFrame, D3DXMATRIX *pParentMatrix,
                   IDirect3DDevice9 *pDevice )
{
    D3DXMATRIX  mtxCombined;

    if ( pParentMatrix != NULL)
      D3DXMatrixMultiply( &mtxCombined, &pFrame->Matrix, pParentMatrix);
    else
      mtxCombined = pFrame->Matrix;

    pDevice->SetTransform ( D3DTS_WORLD , &mtxCombined );
    pFrame->pMesh->Render( );

    // Move on to sibling frame
    if(pFrame->pSiblingFrame)
        DrawHierarchy(pFrame->pSiblingFrame, pParentMatrix, pDevice);

    // Move on to first child frame
    if(pFrame->pChildFrame)
        DrawHierarchy(pFrame->pChildFrame, &mtxCombined, pDevice);
}
```

There are a few key points to note about this code. First, we only need to call this recursive function once from our application, passing in the hierarchy root frame as the first parameter and NULL as the second parameter (because the root frame has no parent). Actually, this latter point is only true if we do not wish the root frame to move in our world in some way. If we wanted to translate the root frame, and thus the entire hierarchy, to some new location in the world, we can pass in a translation matrix as the second parameter. This represents a translation relative to the world frame of reference, which is the root frame's parent under normal circumstances.

Of course, we could simply modify the root frame matrix directly to generate the same result, but it is preferable not to do this. If we change the contents of the root frame matrix (which is virtually always an identity matrix), we make it more difficult to attach this hierarchy to a node in another hierarchy should we want to do so at a later time. For example, consider a machine gun hierarchy that contains moving parts. If we wanted to attach the root frame of the machine gun (assume the handle/grip is the root frame location in the gun model hierarchy) to a character's hand frame, then all we would have to do is make the machine gun root frame a child of the hand frame. If we kept the gun root frame matrix an identity matrix, then as the character moves his hand and passes those transforms down the tree, the matrix multiplication step would just give the machine gun root frame the same position/orientation as the character's hand (since the multiplication by identity just results in the same matrix). Thus attaching and detaching hierarchies to/from other hierarchies to build more complex hierarchy trees becomes a simple matter of pointer reassignment in a linked list.

However, to keep things simple, in the root node case for this example, pParentMatrix will be NULL. Therefore, the matrix of the root frame would not be combined and would instead be used directly as the world matrix for rendering the car body.

You should also note that it is possible for a single frame to contain more than one mesh, so we might imagine that the pMesh pointer actually points to a linked list of meshes attached to this frame. Thus calling the pMesh->Render function would actually step through all sibling meshes in the list and render them. Generally, artists will not design a model where several meshes belong to a single frame because we would have no individual control over these meshes since they would all share the same frame matrix. Applying an animation to the frame matrix would animate all the meshes that are stored in that frame, which is not something we usually want to do. In such an instance, it would probably make more sense for the artist to combine the multiple meshes into a single mesh and assign it to the frame as a single mesh. Either way, for now we can forget about the semantics of multiple meshes per frame because our example deals with only one mesh per frame.

As the code shows, after the child meshes of the current frame have been rendered, it is time to process any sibling frames this level in the hierarchy might have. As it happens, the root frame is the only frame in this level of the hierarchy and does not have any sibling frames. The frame's pSiblingFrame pointer would be set to NULL in this instance and so we move on.

Child frame processing is next. The root frame's pChildFrame pointer will point to Frame1 (in our example). So we call DrawHierarchy again (recursively) passing in as the first parameter the child frame (Frame1) and the root frame matrix as the second parameter (the parent matrix). Keep in mind that at this point the car body mesh has been rendered and the root frame has been totally processed.

As we enter DrawHierarchy for the second time, we are now dealing with Frame1 as the current frame and the root frame matrix is the parent matrix passed in as parameter two. We multiply the frame matrix with the parent matrix so that we now have a temporary combined matrix that contains the complete world space transformation for any meshes in this frame. This combined matrix is sent to the device as the current world matrix and the mesh attached to this frame is drawn (the Wheel 1 mesh in our example).

Next we test to see if this frame has a sibling frame, and indeed it has: Frame1's pSiblingFrame pointer will be pointing to Frame2. So we call the DrawHierarchy function again, passing in Frame2 as the first parameter and the parent matrix passed into this function as the second parameter.

This is a very important point to remember when processing siblings -- we do not pass in the combined matrix that we have just calculated because it is only relevant to the current frame and its children. We can see for example, that while Frame2 should be combined with the root frame matrix (its parent), it should not be combined with Frame1's matrix. So for each sibling that we process, we pass in and use the same parent matrix that was passed into that level of the hierarchy by the level above it. The root frame passed its matrix down to level 2 when it processed its child pointer; now this same matrix (the root frame matrix) is passed along to every sibling at that level and combined with their frame matrices to create their world space transforms.

The story obviously changes when processing child frames. When we call DrawHierarchy for a child frame, we are actually stepping down to the next level of the hierarchy for that particular frame. Therefore, any frames in the lower levels of that branch of the tree must have their matrices combined with the combination of matrices that went before them. If our Wheel 1 frame had a single child frame introducing a new level (a 3rd level) into the hierarchy, this new frame would be traversed down into

when we call DrawHierarchy for the child pointer of Frame1. In that case, we would pass the combined matrix (Frame1 * Parent), and not the initial parent matrix that is used to process siblings. The parent matrix passed into this child of Frame 1, would be (Frame1 Matrix * Root Matrix) which describes the world space position of its parent (Frame 1). It is this world space position that is the frame of reference for the matrix that would be stored in Frame 1's child. In summary, the process is as follows:

1. **For each frame**
2. **Combine frame matrix with parent matrix (if exists)**
3. **Render current frame's mesh(es) using combined matrix**
4. **Process all sibling frames (frames on the same level of the hierarchy). Pass them the same parent frame that the current frame was passed, so that they too can combine their matrices with the parent matrix.**
5. **Process the child frame of this frame. Pass in the new combined matrix that was used to render the mesh(es) at this frame. This effectively steps us down into the next level of the hierarchy for children of the current frame where the world space position generated for the current frame, becomes the frame of reference that any child frames have been defined relative too.**

Hopefully the concept of a spatial hierarchy now makes more sense to you. It is vital that you understand how these relationships work because we will encounter them again and again in graphics programming projects. Not only will we be dealing with spatial hierarchies here in this chapter, but we will see them again in the next two chapters and indeed throughout this entire series of courses. Make sure that you are comfortable with how a hierarchy works in theory as well as how to traverse and render one. In summary, we can think of a spatial hierarchy as a nested set of local space coordinate systems, where each local system is defined relative to the parent system it is attached to one level up in the tree. Also, do not forget that changes to a frame's matrix will automatically influence all of the world matrices generated for its children, since they are defined in its local space.

9.2 Frame Animation

To apply animation to a given frame, the most common approach is to assign it a unique **animation controller**. The animation controller's job is to ensure that the frame matrix to which it is attached is properly updated for the frame (i.e. it *controls* how the frame animation proceeds). Note that animations are generally going to be locally applied to a given frame (which in turns affects any children of course). In this section we will compose some simple code to help examine this process. To be clear, this will all be for illustration only -- we will not use any of this code in our applications. We will examine animation in much more detail in the next chapter, but for now, let us just get a high level understanding of the process since it illustrates the need for frame hierarchies and lets us see how they work.

Let us imagine that we wanted to rotate the four wheels of our example car by some fixed amount over time. We can do this by periodically applying transformations to the matrices of the wheel frames. In our example, each wheel frame requires local rotation updates at regular intervals and thus four animation controllers will be needed; one to manage each frame matrix. For the sake of this discussion, let us call this animation controller type a CAnimation. We will see later how this fits in with the D3DX

framework we will be using and thus we are going to get used to this naming convention early on. Our animation controller class (CAnimation) in this case could be very simple since it need only rotate the wheels about their center points.

```
class CAnimation
{
public:
    CFrame *pAttachedFrame;
    void Rotate ( float Degrees ) ;
};
```

Our CAnimation object is tasked with applying rotations to a single frame matrix. It contains a pointer to a frame in the hierarchy (i.e, the frame which it should animate) and a method that will build a rotation matrix to concatenate with the frame matrix whenever it is called. Calling this function multiple times will allow us to accumulate rotations for the frame by a specified number of degrees.

After the frame hierarchy is loaded at application startup and stored in memory as a tree of D3DXFRAME structures (we will see how to do this later in the chapter), we can call a function that searches through the hierarchy for the wheel frames and creates a CAnimation instance for each one. Again, our CAnimation object is a controller, attached to a single frame and used to manipulate its matrix to animate the frame (and thus its children). In a moment we will see how a global function might be constructed to traverse the frame hierarchy searching for the wheel frames and create new CAnimation objects which will be attached to each of them. For each animation, we will set its pAttachedFrame pointer to point at the wheel frame it is intended to animate.

> **Note:** When we use the D3DX library later to load a frame hierarchy from an X file, we will see that each frame will include a string storing its name. This is typically assigned to the frame by the artist or modeler who created the X file. This allows us to locate a frame in the hierarchy by searching for its name. In our hierarchy we can imagine that frames 1, 2, 3 and 4 would have the names 'Wheel 1', 'Wheel 2' , 'Wheel 3' and 'Wheel 4'. In the example code that follows we use the _tcsstr function to search for all frames that have the words 'Wheel' in their frame name.

As our four CAnimation objects will, in effect, combine to create a set of animation data for the automobile we will also write a simple container class that will handle the task of maintaining the list of animations. We will appropriately call this container a CAnimationSet. In our example, we will need to add four CAnimation objects to our CAnimationSet because we wish to animate the four wheels of our automobile (thus we wish four animations to be played).

```
class CAnimationSet
{
 public:
   CAnimation   **m_ppAnimations;
   DWORD          m_nNumberOfAnimations;

   void AddAnimation ( CAnimation pAnimation );
};
```

Once the X file data is loaded into the hierarchy we will call our CreateAnimations function (see below) to search for the four wheel frames and create, attach, and store our CAnimation objects. CreateAnimations would be recursive of course and need only be called once by our application. The

application will pass in a pointer to the root frame of the hierarchy to start the process, along with a pointer to an empty CAnimationSet object. Each CAnimation created by this function will be added to the CAnimationSet object.

```
void CreateAnimations ( CFrame *pFrame , CAnimationSet *pAnimationSet )
{
    CAnimation *pAnimation = NULL;
    if ( !pFrame )  return;
    if ( _tcsstr ( pFrame->pFrameName , "Wheel" )
    {
       pAnimation = new CAnimation;
       pAnimation->pAttachedFrame = pFrame;
       pAnimationSet->AddAnimation ( pAnimation );
    }

    if ( pFrame->pSiblingFrame ) CreateAnimations (pFrame->pSiblingFrame);
    if ( pFrame->pChildFrame   ) CreateAnimations (pFrame->pChildFrame);
}
```

When the above function returns, the application's CAnimationSet object will have an array of four pointers to CAnimation objects (one for each wheel frame) and its m_nNumberOfAnimations member will be set to 4. Notice that the CreateAnimations function adds a newly created animation to the animation set using the CAnimationSet::AddAnimation member function. We will not show any code for this function but we can imagine that it might be a simple housekeeping function that resizes the m_ppAnimations array and adds the passed CAnimation pointer to the end of it.

Once our main game loop is running, we will need to animate and render the hierarchy. The animation update might take place in a separate application-level animation function (ex. AnimateObjects) or it could take place as we traverse the hierarchy for rendering. If we were going to use the latter approach, then each frame would have to know about its assigned controller so that it could run its update on the spot, prior to concatenating with its parent matrix. If we are going to do the animation updates before the hierarchy traversal, then this is not necessary. For example we could write a function that loops through all of the animations in the animation set and calls their Rotate functions like so:

```
void AnimateObjects ( float AngleInDegrees )
{
      for ( DWORD i = 0 ; i < pAnimationSet->m_nNumberOfAnimations; i++ )
      {
            CAnimation *pAnimation = pAnimationSet->m_ppAnimations[i];
            pAnimation->Rotate( AngleInDegrees );
      }
}
```

This function would ensure that all of the frame matrices were updated by their attached animation controllers prior to the hierarchy traversal. The CAnimation::Rotate method would be responsible for applying a local rotation for the frame to which it is attached. Again, this will ultimately wind up rotating all child frames and child meshes as well, once we do our hierarchy traversal.

In our example it is assumed that each wheel mesh has been defined in its own model space such that its center is at the origin and it is facing down the positive Z axis. We can think of the model space X axis

as being the axle on which the wheel rotates. Thus, we wish the animation object to rotate its attached wheel around the wheel frame's local X axis. In this case, we might imagine our CAnimation::Rotate method to look something like this.

```
void CAnimation::Rotate ( float Degrees )
{
   D3DXMATRIX  mtxRotX;
   D3DXMatrixRotationX( &mtxRotX , D3DXToRadian (Degrees) );
   D3DXMatrixMultiply(&pAttachedFrame->Matrix, &mtxRotX, &pAttachedFrame->Matrix);
}
```

As you can see, the function will first build an X axis rotation matrix. This will then be multiplied with the relative matrix of the attached frame, the result of which will replace this frame matrix. The order of the matrix multiplication is significant. We wish to rotate the wheel about its own X axis and then combine it with the current matrix of the frame so that the rotated wheel is then translated out to its position relative to the parent frame *after* being rotated. Therefore, we do (RotMatrix * FrameMatrix) so that the final matrix first rotates any children around their local X axis and then positions them relative to the parent frame of reference.

In our current example, each time the Rotate function is called, the matrices belonging to the wheel frames will store cumulative rotations. That is, the rotation is permanent and the frame matrix contents are directly modified (not done in a temporary matrix) by the attached CAnimation object. This may or may not be desirable depending on the circumstances, but it illustrates the point and works fine.

Rendering the newly animated hierarchy is no different than the code we saw previously. All our CAnimation objects have done is modified the relative matrix stored in the frame for which each is attached. We still start at the root and work our way down through the levels, combining the matrices of each frame with their parent. When we reach the frame for wheel 1, the parent matrix will be combined with the newly rotated frame matrix to create the frame world matrix and the wheel will be rendered. It will still be in its correct position relative to the car (wherever we may have moved the root frame to in the world) but will have been successfully rotated about its own center point.

Again you are reminded that these localized rotations affect all children for a frame – meshes and other child frames alike. For example, imagine that our car representation has another frame containing the mesh of a wheel hubcap attached to wheel 1 as a child (Fig 9.2).

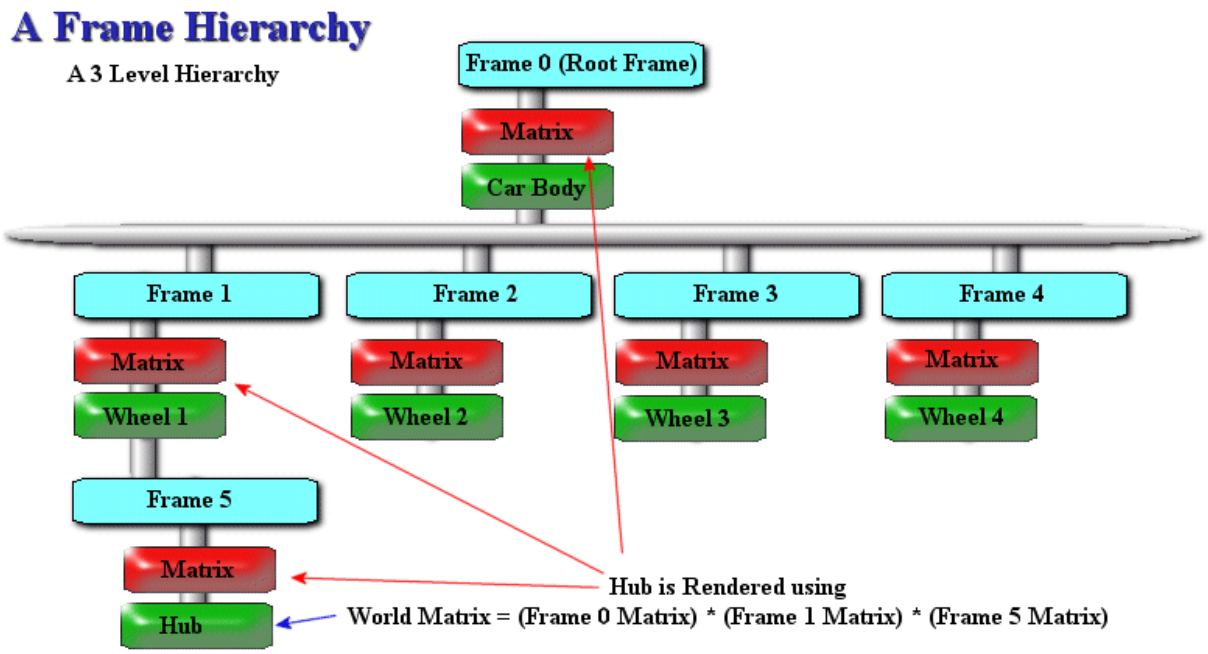


Figure 9.2

When we arrive at Frame 5 in our tree, the DrawHierarchy function will have been passed a parent matrix from the previous recursion (when processing Frame 1). That matrix will actually be the root frame matrix combined with the Frame 1 matrix. If the animation attached to Frame 1 rotated its assigned frame matrix, then this rotation will be propagated down to the hubcap frame as well and the hubcap will rotate along with its parent. This is as it should be of course since the hubcap is defined in the (now modified) local space of its parent.

As we will see later in this course, it is precisely this characteristic of a hierarchy that makes them perfect candidates for modeling articulated structures like game characters. It is easy to see the hierarchical relationship between the parts of the human body. The upper arm (origin = shoulder) would have the forearm (origin = elbow) as a child frame. The forearm would have the hand (origin = wrist) as its child frame and the hand would have five finger frame children (which might themselves be jointed and linked in a similar manner). If we wanted the model to raise its arm in the air by rotating about the shoulder, we could simply have a CAnimation object attached to the upper arm frame generate a local rotation matrix and use it to transform the frame (by multiplying it with the upper arm frame matrix). As a result, all of the children will move as expected since the rotation is propagated down through the hierarchy during our tree traversal.

So in summary, we now understand what a hierarchy is and we should be to see how an artist might design a scene or an object, not as a single mesh, but as a number of different meshes which are all spatially positioned and oriented relative to one another. These meshes are stored in frames, where each frame will have a transformation matrix that will not represent an absolute world transformation but a relative transformation describing the relationship of its own local system with its parent frame. The exception to the rule is the root frame which initially has no parent. The root frame matrix can instead be thought of as storing the core model space coordinate system (i.e. the base reference frame) from which

the rest of the hierarchy proceeds. This means that it can serve as a world space transformation for the entire hierarchy if it takes on values other than identity.

We also know that when we render the hierarchy we must process each frame, combining its matrix with the current combined matrix containing the concatenation of all the higher level frames in the hierarchy linked via its parent. We examined how to render the siblings and child frame objects for each frame using the parent matrix and the combined frame matrix respectively. Finally, we learned that because the matrices for our frames are not absolute, we can very easily animate individual frames in the hierarchy in their own local space and watch those animations propagate down through the subsequent levels of the tree. We used the concept of animation objects (CAnimation) to handle the animation (matrix updating) of the frame to which it is attached.

Finally, we discussed that a set of frame animations could be intuitively packaged together to represent an animation set. While the discussion of CAnimation and CAnimationSet classes may have seemed trivial since the animation data could be contained in any arbitrary structures, getting used to the idea of how these objects work with the frame hierarchy will greatly help us in understanding the loading of multi-mesh X files. It will also help us understand the animation system that D3DX provides to assist with the animation of frame hierarchies.

Unfortunately, when we load an X file using D3DXLoadMeshFromX or its sibling functions, any hierarchy information stored in the file is discarded. Instead, the function traverses the hierarchy in the file, applying the relative transformations so that all of the vertices in all of the meshes in the file are now in their model space positions. At this point, the mesh information has now been correctly transformed and collapsed into a single mesh object which is returned to the caller. So in the case of our automobile hierarchy, D3DXLoadMeshFromX would correctly offset each of the wheels from the car body mesh, but then collapse it all into a single mesh object, discarding the entire hierarchy. While the single mesh will look correct when not in motion, because the wheels are not separate meshes, we have lost the ability to animate them independently from the rest of the car.

Not to worry though, we do have some solutions at our disposal that allow for preservation of the hierarchy. As this chapter progresses, we will begin to examine some of our options. To get things started, the best place to begin is by looking more closely at the way our data is going to be stored. So in the next few sections we will examine X files in some detail.

9.3 X File Templates

Before DirectX 9.0, hierarchical scenes stored in X files had to be loaded and parsed manually using a set of packaged COM objects. These COM objects still exist today and they can continue to be used to parse X files, but as of DirectX 9.0, the D3DX library contains a function called D3DXLoadMeshHierarchyFromX that automates the entire process. The complete hierarchy is loaded (all child frames, meshes, textures, materials, and so on) in such a way that it still affords our application complete control over how the hierarchy data is allocated in memory. This level of insulation is certainly welcome and it will save us a good deal of coding and debugging, but it is still helpful to be at least somewhat familiar with how the data is stored in the X file and how it can be extracted. That is the purpose of this section.

The X file format is essentially a collection of templates. A **template** is a data type definition, very similar to a C structure, describing how the data for an object of a given type is stored in the file. Just as a C structure describes how variables of that type will have their member variables arranged in memory, an X file template describes how data objects with the same name (and GUID) will have their data laid out in the file. Once we know how to inform DirectX about the components of a template, the X file loading functions will recognize objects of that type in the file, know exactly how much memory is required, and understand how to load the data. The X file format is actually similar to the IWF format in many ways. We can think of a data object in the X file as the equivalent of the IWF chunk. Just as there are many different chunk types in the IWF format for representing different types of information, there are different data objects in an X file represented as templates. IWF chunks can also have child chunks, just as X file templates can have child templates for data objects.

For example, the X file Frame template can have many different types of child templates: the TransformationMatrix data type (which is the matrix for a frame in the hierarchy) and the Mesh data type (of which there can be many) are common examples. The Frame can also have other Frames as children, which we now know is the means by which a hierarchy is constructed.

As in the IWF format, you can also create custom data templates that are understood by your own application, but are not a part of the X file standard. For example, you could have a PersonalDetails template which contains a name, address, and telephone number, turning the X file into a standard address book (not that we would likely do such a thing). When the D3DXLoadMeshHierarchyFromX function encounters a custom data object, it will pass the custom data to your application using something similar to a callback function. This gives the application a chance to process that data.

Fortunately, we will not have to create our own templates to load a hierarchical X file. When using the D3DX library supplied X file loading functions, the standard templates are registered automatically. We will be studying some of the standard templates in this lesson to get a feel for how a hierarchy is laid out in the X file, but for details on all standard templates (Table 9.1), look at the SDK documentation in the section *DirectXGraphics / Reference / X File Reference / X File Format Reference / X File Templates*.

Table 9.1: X File Standard Templates

• Animation • AnimationKey • AnimationOptions • AnimationSet • AnimTicksPerSecond • Boolean • Boolean2d • ColorRGB • ColorRGBA • Coords2d • DeclData • EffectDWord • EffectFloats • EffectInstance • EffectParamDWord • EffectParamFloats • EffectString • FaceAdjacency • FloatKeys • Frame • FrameTransformMatrix • FVFData • IndexedColor	• Material • Matrix4x4 • Mesh • MeshFace • MeshFaceWraps • MeshMaterialList • MeshNormals • MeshTextureCoords • MeshVertexColors • Patch • PatchMesh • PatchMesh9 • PMAttributeRange • PMInfo • PMVSplitRecord • SkinWeights • TextureFilename • TimedFloatKeys • Vector • VertexDuplicationIndices • VertexElement • XSkinMeshHeader

Note: We will not need to register these standard templates with DirectX. In fact, even if we use custom objects in our X file which are not part of the standard, we do not have to register those either. We can simply store the template definitions at the head of the X file itself (just underneath the standard X file header) and the Direct X loading functions will automatically register them for us before it parses the file. More on this momentarily...

9.3.1 X File Components

At the top of every X file is a small line of text which looks something like this:

Xof 0303txt 0032

This value is written to the file by the modeling application and serves as a signature to identify the file to the loading application as a valid X file. The ‘Xof’ portion identifies it as an X file. The ‘0303’ text tells us that this X file was built using the X file version 3.3 templates. Following the version number is ‘txt’, which tells us that this file is a text file. X files can also be stored in binary format, and if this is the case then we would see ‘bin’ instead. Finally, the file signature line ends with ‘0032’ which tells us the file uses 32-bit floating point values instead of 64-bit floats.

We never have to read or parse this line manually as it is handled by the D3DX loading functions and the appropriate action will be taken. The following example shows the signature of a binary X file created using the version 3.2 templates with 64-bit floating point precision:

Xof 0302bin 0064

Following the signature there may be a list of template definitions that describe the data format for objects found in the file. This will typically be the case only if the X file uses custom data objects, but it is possible that some X files may include template definitions that are part of the standard.

X files contain three types of data objects: **top level**, **child**, and **child data reference**. A data object is a packet of information stored in the format of a registered template. Let us have a look at a very simple X file to see what a data object is and how it is stored. This example will use a common top level data object: a Mesh. The Mesh will contain a child data object for defining a Material. First we will look at the DirectX standard templates for both a Mesh and a Material.

```
template Mesh
{
        <3D82AB44-62DA-11cf-AB39-0020AF71E433>
        DWORD   nVertices;
        array   Vector vertices[nVertices];
        DWORD   nFaces;
        array   MeshFace faces[nFaces];
        [ ... ]                         // An open template
}
```

The template above is of type ‘Mesh’ and this is the name we would use to define instances of this object in the X file. The first member of any template is its GUID. Every template must include a GUID so that it can be uniquely identified. This prevents name mangling when a custom template shares the same name as another custom template or one of the standard templates. Keep this requirement in mind if you develop your own custom templates.

You can generate a GUID using the ‘Guidgen.exe’ tool that ships with the Microsoft visual C/C++ compiler. You can find it in the ‘\Common\Tools’ directory of your MSVC installation. Fig 9.3 shows the tool in action.

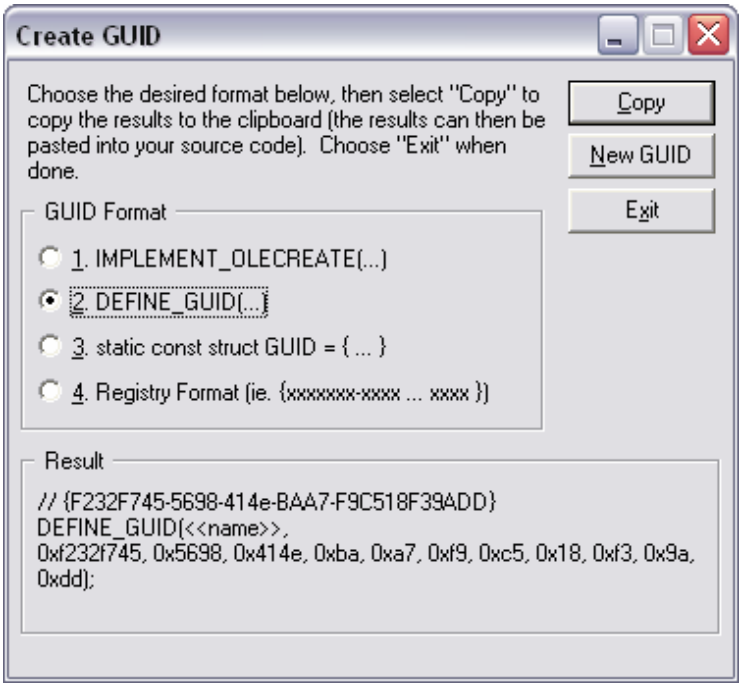


Figure 9.3

Guidgen uses a combination of parameters (hardware serial numbers, current date/time, etc.) to generate an ID that is guaranteed to be unique throughout the world. If for some reason you do not like the GUID it generates, you can simply hit the 'New GUID' button to build another one. In Fig 9.3 the second radio button selected helps format the GUID in the result window so that you can cut and paste it into an application. If we were to generate a GUID for a custom template called Contact, we would include it in the definition like so:

```
template Contact
{
     < F232F745-5698-414e-BAA7-F9C518F39AD >
     STRING Name;
     STRING Address1;
     STRING Address2;
     STRING PhoneNumber;
     DWORD Age;
}
```

Here we have simply copied over the top line of the Guidgen result pane and removed the two forward slashes and replaced the curly braces with chevrons.

When we use the D3DXLoadMeshHierarchyFromX function to load an X file, all standard templates will be recognized by their GUIDs and loaded on our behalf. In the case of a custom data object, something more akin to a callback function will be used instead. The application defined callback function (which will be triggered by D3DX when a custom template is encountered) will check the GUID that is passed and determine whether or not it matches a custom template the application would like to support. The application can skip this data if it does not match.

The DEFINE_GUID macro stores GUID information in a structure with a name that you can assign so that you can do fast GUID comparison tests. For our Contact template example above:

```
// {F232F745-5698-414e-BAA7-F9C518F39ADD}
DEFINE_GUID(ID_Contact, 0xf232f745, 0x5698, 0x414e, 0xba, 0xa7, 0xf9, 0xc5, 0x18,
0xf3, 0x9a, 0xdd);
```

This might look like a complex way to assign a name to a GUID at first, but if you examine the Guidgen result window, you will see that the code is the same. We have simply substituted ID_Contact in the <<name>> section.

With this alias defined, our application callback mechanism can do a quick test to determine whether to process a data object passed from a D3DX loading function:

```
if( *pDATAGUID == ID_Contact ){ // Process the objects data here }
```

Again, for standard templates, these steps are unnecessary. All standard templates such as the Mesh template we are discussing will be identified automatically by D3DXLoadMeshHierarchyFromX and loaded on our behalf.

Let us take another look at the standard Mesh template and continue our discussion of its members.

```
template Mesh
{
        <3D82AB44-62DA-11cf-AB39-0020AF71E433>
        DWORD   nVertices;
        array Vector   vertices[nVertices];
        DWORD   nFaces;
        array MeshFace faces[nFaces];
        [ ... ]                         // An open template
}
```

The basic building blocks for a template are the data types that it stores. For example, we see that the first data member after the GUID in this template is a DWORD which describes how many vertices are in the mesh. Table 9.2 presents a list of all of the basic data types you can use in a template.

Table 9.2: Template Data Types

Type	Size
WORD	16 bit integer
DWORD	32 bit integer
FLOAT	32 bit floating point value
DOUBLE	64 bit floating point value
CHAR	8 bit char
UCHAR	8 bit unsigned char
BYTE	8 bit unsigned char
STRING	NULL terminated string

We can also specify arrays of the supported types or even other templates using the **array** keyword. A good example is the array of Vector objects in the mesh which contains one Vector for each mesh vertex. The Vector template is another standard template:

```
template Vector
{
   < 3D82AB5E-62DA-11cf-AB39-0020AF71E433 >
    float x;
    float y;
    float z;
}
```

The preceding nVertices DWORD (in the Mesh template) describes the number of items in the vector array that follows. This is the pattern that will be followed when arrays are used in a template. In this example it describes the number of vertices the Mesh object contains followed by an array of that many Vector data objects.

Following the Mesh vertex array is another DWORD that describes how many faces are in the Mesh. That value is followed by an array of MeshFace objects (another standard template).

```
template MeshFace
{
   < 3D82AB5F-62DA-11cf-AB39-0020AF71E433 >
    DWORD nFaceVertexIndices;
    array DWORD faceVertexIndices[nFaceVertexIndices];
}
```

This object type describes a single face in a mesh. The first member is a DWORD describing how many vertices the face uses (3 for a triangle, 4 for a quad, etc.) and it is followed by an array of that many DWORDs. Each DWORD in the array is the **index** of a vertex in the Vector array just discussed.

All of these components describe the Mesh object in its most basic form and every Mesh object must have the previously discussed components (a vertex and face list). At the bottom of the Mesh template we have a set of square brackets with ellipsis in between. This indicates that the Mesh template is an *open template* format that can include any type and any number of additional data objects.

9.3.2 Open, Closed, and Restricted Templates

Some templates will include child objects, and others will not. Templates that cannot include children are considered **closed**. This is the default template type. Conversely, templates that allow for any child type and number are called **open** templates. The Mesh template is an example of an open template. The third type of template can have child objects, but only of specific types specified by the template. These are called **restricted** templates and they must specifically list the templates that are legal child objects. A restricted template can have any number of the data objects specified by the template as embedded child data objects.

When a template is closed it cannot have any children. If children are somehow included in a closed template, they will be ignored by the loading code. The following example will demonstrate what this means. We start with two example templates:

```
template Matrix4x4 {
  <DED0E885-ED92-4224-A090-E81FC8AB2B83>
  array FLOAT Matrix[4][4];
}

template TestTemplate {
  <A96F6E5D-04C4-49C8-8113-B5485E04A866>
  DWORD Flags;
}
```

At the moment, these are closed templates. But we can certainly imagine a situation where an object like the following could be useful:

```
TestTemplate ChildTest {
  0;                    // Flags
  Matrix4x4 MyMatrix    // Matrix
  {
      1.0; 0.0; 0.0; 0.0;
      0.0; 1.0; 0.0; 0.0;
      0.0; 0.0; 1.0; 0.0;
      0.0; 0.0; 0.0; 1.0;;
  }
}
```

Given the previous template definitions, this is not legal. Technically we can do it, but the D3DX loading functions would completely ignore the matrix data because the TestTemplate definition described a closed data object which cannot have child objects (the Matrix4x4 object above) embedded. However, if we were to use ellipsis in the template definition to open the template, our usage above becomes completely legal:

```
template TestTemplate {
  <A96F6E5D-04C4-49C8-8113-B5485E04A866>
  DWORD Flags;
  [ ... ]
}
```

You may have noticed the additional semi-colon at the end of the child matrix data object in the above example. This was not a typo -- it signifies the end of a child object within a parent data object. Therefore, we have a semi-colon separating each member of the matrix plus an additional semi-colon denoting the end of the matrix data object within the outer data object. There is some helpful discussion of the usage of commas and semi-colons in the SDK documentation.

The use of the ellipsis is much like their use in C/C++: they indicate that some variable set of data can be included. To restrict the legal child types for a template (therefore creating a restricted template type), we simply include the list of types within the square brackets:

```
template TestTemplate {
  <A96F6E5D-04C4-49C8-8113-B5485E04A866>
  DWORD Flags;
  [ Matrix4x4, Vector3 ]
}
```

The comma separated list for a restricted template can include type names with optional GUIDs:

```
[ Matrix4x4 <DED0E885-ED92-4224-A090-E81FC8AB2B83>, Vector3 ]
```

Using a GUID is not required, but it can be a good idea to avoid ambiguity if new templates are introduced that cause a name clash. It also provides the ability to overload templates with the same name and restrict children to a specific type.

Describing the X file format layout is not easy and often is best accomplished with a simple example of an actual X file.

X File Sample 1: Basic Mesh

Here we see the contents of an actual X file that contains a single top level data object (a Mesh). Notice how the data is laid out in the exact format described by the Mesh template discussed earlier.

```
Xof 0303txt 0032

mesh{
  6;              // Number of vertices
 -5.0;-5.0;0.0;,  // Vertex list ( Vector array )
 -5.0;5.0;0.0;,
  5.0;5.0;0.0;,
  5.0;-5.0;0.0;,
 -5.0;5.0;-5.0;,
 -5.0;5.0;5.0;;

  2;              // Number of faces
  3;0,1,2;,       // MeshFace Array
  3;3,4,5;;
}
```

This example contains no color information or vertex normals, but it does define a basic mesh and provide the core requirements discussed previously.

Since the Mesh template is open, we can embed any child data objects we wish, although only a certain subset of child data objects will be understood by the D3DX loading functions. So let us take a look at another standard X file template that defines a material just like the D3DMATERIAL9 structure we looked at in Chapter Five of Module I.

```
template Material
{
  < 3D82AB4D-62DA-11cf-AB39-0020AF71E433 >
    ColorRGBA  faceColor;
    float      power;
    ColorRGB   specularColor;
    ColorRGB   emissiveColor;
}
```

Notice that the standard Material template does not include an Ambient member. We actually mentioned this in passing in Chapter Eight, and now we see this to be true at the low level. The D3DXLoadMeshFromX function could not return ambient information for each material used by the mesh being loaded because the material data objects used to describe mesh materials in the X file have no ambient member.

The faceColor member of the Material template is equivalent to the Diffuse color member of the D3DMATERIAL9 structure. This is represented using the standard template ColorRGBA seen next. The specular and emissive members use the standard template ColorRGB, also shown next. We are already familiar with how the DirectX lighting pipeline deals with materials and, with the exception of the missing ambient component, the X file material data object describes material properties in the same way as its D3DMATERIAL9 counterpart. It is the data stored within these data objects that the D3DXLoadMeshFromX function returns in its D3DXMATERIAL array.

The ColorRGBA Template	The ColorRGB Template

`template ColorRGBA`
`{`
`<35FF44E0-6C7C-11cf-8F52-0040333594A3>`
`  float red;`
`  float green;`
`  float blue;`
`  float alpha;`
`}`
	`template ColorRGB`
	`{`
	`  <D3E16E81-7835-11cf-8F52-0040333594A3>`
	`   float red;`
	`   float green;`
	`   float blue;`
	`}`

We will now expand our X file example by adding a material data object as a child of the mesh.

X File Sample 2: Basic Mesh with Material

```
Xof 0303txt 0032

mesh{
    6;              // number of vertices
   -5.0;-5.0;0.0;,  // vertex list
   -5.0;5.0;0.0;,
    5.0;5.0;0.0;,
    5.0;-5.0;0.0;,
   -5.0;5.0;-5.0;,
   -5.0;5.0;5.0;;

    2;            // number of faces
    3;0,1,2;,     // face list
```

```
    3;3,4,5;;

    Material {     // child data object of mesh
      1.000000;1.000000;1.000000;1.000000;;
      0.000000;
      0.000000;0.000000;0.000000;;
      0.000000;0.000000;0.000000;;
    }
}
```

Here we see a top level object (the Mesh) with one child (the Material). The mesh is considered a top level object because it has no parent. That is to say, it is not embedded in any other data object.

Defining materials as direct children of objects like this means that this material object belongs to, and can only be referenced by, this mesh. While this is sometimes the case, most of the time X files will represent materials (and other children) as **data reference objects**. A data reference object is used to save space. If the X file contained five meshes that all required the same material, we can save space by creating the material once at the top of the X file as a top level data object and assigning it a name (a reference). All meshes that use the material can use the name as a reference, much like the concept of instances we looked at in previous lessons. This is obviously much more memory space efficient than defining the same material data object in each of the five separate meshes.

This next example uses the same basic X file, but this time the material is defined as a top level object and assigned a name so that any number of meshes in the file would be able to reference it.

X File Sample 2: Basic Mesh with Material Reference

```
Xof 0303txt 0032

Material WhiteMat {     // This is a Data Object
   1.000000;1.000000;1.000000;1.000000;;
   0.000000;
   0.000000;0.000000;0.000000;;
   0.000000;0.000000;0.000000;;
}

mesh{                   // This is a Data Object
    6;                  // number of vertices
   -5.0;-5.0;0.0;,      // vertex list
   -5.0;5.0;0.0;,
    5.0;5.0;0.0;,
    5.0;-5.0;0.0;,
   -5.0;5.0;-5.0;,
   -5.0;5.0;5.0;;
    2;             // number of faces
    3;0,1,2;,      // face list
    3;3,4,5;;

   {WhiteMat}     // Data Reference Object as a child of the mesh Data Object
}
```

We can use any top level data object as a child of another object (assuming we have given the data object a name) by placing the name in between two curly braces. In the previous example, the line '{WhiteMat}' signifies that WhiteMat is the first material used by this mesh. D3DX supplied loaders automatically handle cases for both child objects and child reference objects. However, the same rules still apply for closed and restricted templates. References will only be valid in open templates or if explicitly included in the type list for a restricted template.

In the previous examples, we saw both a material data object and a material data reference object being embedded as a direct child of the mesh object. While this was a nice illustration of the concept, most meshes will not assume either form. Meshes will often use multiple materials (recall the subset discussion from Chapter Eight), and they will need to be mapped to faces in the mesh. So instead, there will be a single material list data object stored inside each mesh and the material objects (or material references) will be stored inside this object. The MeshMaterialList standard template contains a list of materials used by the mesh as well as the face mapping information:

```
template MeshMaterialList
{
    < F6F23F42-7686-11cf-8F52-0040333594A3 >
        DWORD nMaterials;
        DWORD nFaceIndexes;
        Array DWORD FaceIndexes;
       [Material]
}
```

The first DWORD describes how many Material data objects are embedded as child data objects or child data reference objects of this material list object. This is followed by the number of faces the material list affects. This will usually be equal to the number of faces in the entire face list of the mesh unless there is only one material and then nFaceIndexes=1 (indicating that this single material should be applied to all faces in the mesh). Following this is an array of DWORDS describing which material is applied to which face. Each element in this array is a zero-based index describing a material from a list of child materials that will be embedded in this object. This is a restricted template that can only have Material objects as children.

X File Sample 3: Basic Mesh with Reference Based MeshMaterialList

Our next example will embed a MeshMaterialList in a Mesh object and the material list will have two Material children. We can see the way that X files work using a hierarchy of sorts, even when it is not a spatial one. Also note that the Mesh and the Materials are top level data objects and that we are **referencing** the materials from within the MeshMaterialList.

```
Xof 0303txt 0032

Material WhiteMat{  // Top Level Data Object
   1.000000;1.000000;1.000000;1.000000;;
   0.000000;
   0.000000;0.000000;0.000000;;
   1.000000;0.000000;0.000000;;
}
Material RedMat{    // Top Level Data Object
   0.000000;0.000000;1.000000;1.000000;;
```

```
   0.000000;
   0.000000;0.000000;0.000000;;
   0.000000;0.000000;1.000000;;
}
mesh{               // Top Level Data Object
   6;               // Number of vertices
  -5.0;-5.0;0.0;,   // Vertex list
  -5.0;5.0;0.0;,
   5.0;5.0;0.0;,
   5.0;-5.0;0.0;,
  -5.0;5.0;-5.0;,
  -5.0;5.0;5.0;;
   2;               // Number of faces
   3;0,1,2;,      // Face list
   3;3,4,5;;      // Second Face

   MeshMaterialList {   // child data object of Mesh
     2;                 // Two materials in the list
     2;                 // Two faces in the following material face list

     0,                 // the first face uses Material 0
     1;;                // the second face uses Material 1

     {WhiteMat}
     {RedMat}

   }// End Material list

}// end of mesh object
```

As we can see here, the mesh has as a child data object: the MeshMaterialList data object. As specified by the template, its first value describes the number of materials that will be assigned to this list (2). This is followed by the number of faces in the mesh which will have materials assigned to them (two in this example). Following this information is the index data that describes for each face, which material it uses in the list that follows. The first face index is zero which describes the first face in the mesh as having ‘WhiteMat’ (Material 0) assigned to it. The second face index is a value of 1 meaning it has been assigned the second material (Material 1) in the list ‘RedMat’. Following the index list is the list of materials belonging to the material list and for which the indices above it are relevant. In this example, the materials are referenced and are actually defined as top level objects at the head of the file. This will allow any other meshes which may also be represented in the file to reference these same materials in their material list data objects, thus saving storage space.

X File Sample 4: Basic Mesh with Reference Based MeshMaterialList

For the sake of completeness, let us see the same example again, but with the materials embedded in the MeshMaterialList data object rather than treated as top level data objects.

```
Xof 0303txt 0032

mesh{
 6;               // number of vertices
-5.0;-5.0;0.0;,   // vertex list
```

```
-5.0;5.0;0.0;,
 5.0;5.0;0.0;,
 5.0;-5.0;0.0;,
-5.0;5.0;-5.0;,
-5.0;5.0;5.0;;
 2;         // number of faces
 3;0,1,2;,  // face list
 3;3,4,5;;


 MeshMaterialList
 {    // child data object of Mesh
   2; // Two materials in the list
   2; // Two faces in the following material face list

       // material face list
   0,  // the first face uses Material 0
   1;; // the second face uses Material 1

   Material { // Material 0,child data object of MaterialList
   1.000000;1.000000;1.000000;1.000000;;
   0.000000;
   0.000000;0.000000;0.000000;;
   0.000000;0.000000;0.000000;;
   }
   Material { // Material 1, child data object of MaterialList
   1.000000;0.000000;0.000000;1.000000;;
   0.000000;
   0.000000;0.000000;0.000000;;
   0.000000;0.000000;0.000000;;
   }
 } // end material list

}// end of mesh object
```

The current example is a fairly complete X file. Indeed, if your meshes do not contain any textures, texture coordinates, or vertex normals, then your meshes might look very similar to this. You should try opening up some text based X files and examining them to get a better understanding of the format and the way that custom data objects have their templates listed at the top of the file.

To continue, let us look at how texture information would be stored for a mesh in an X file. We know from our last chapter that texture pixel data is not stored inside the X file (although you could create a custom template to store image data directly), only the file name is stored. Recall that D3DXLoadMeshFromX passed back a buffer of D3DXMATERIAL structures, where each element contained a D3DMATERIAL9 and a texture file name. We can assume then that in X files, texture and material data can somehow be paired together, and this is indeed the case -- instead of each face having a separate material and texture index, the Material object we just studied will now contain a TextureFilename child object. Material objects can still exist on their own without texture filename objects inside them (as seen previously), but often, a Material will describe a material/texture combination used by faces in the mesh.

The TextureFilename data object is a simple one that contains a string describing the name of the texture image file.

```
template TextureFilename
{
  < A42790E1-7810-11cf-8F52-0040333594A3 >
    string filename;
}
```

Let us now update our example to see what our mesh would look like in X file format if we now used materials that also referenced textures.

X File Sample 5: Materials storing Texture Filenames

```
Xof 0303txt 0032

Material BrickMat{            // Top Level Data Object
1.000000;1.000000;1.000000;1.000000;;
0.000000;
0.000000;0.000000;0.000000;;
1.000000;0.000000;0.000000;;
Texture Filename {“BrickWall.bmp”;}
}

Material WaterMat{            // Top Level Data Object
0.000000;0.000000;1.000000;1.000000;;
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;1.000000;;
TextureFilename {“WaterTexture.bmp”;}
}

mesh{             // Top Level Data Object
  6;              // Number of vertices
 -5.0;-5.0;0.0;,  // Vertex list
 -5.0;5.0;0.0;,
  5.0;5.0;0.0;,
  5.0;-5.0;0.0;,
 -5.0;5.0;-5.0;,
 -5.0;5.0;5.0;;
  2;        // Number of faces
  3;0,1,2;, // Face list
  3;3,4,5;;

  MeshMaterialList {// Child data object of Mesh
   2;               // Two materials in the list
   2;               // Two faces in the following material face list

   0,               // the first face uses Material 0
   1;;              // the second face uses Material 1

   {WallMat}
   {WaterMat}

  }// End Material list
```

```
}// end of mesh object
```

When we look at the Material data objects which contain both color and texture information, we understand that it is exactly this information which the D3DXLoadMeshFromX function returns to our application in the D3DXMATERIAL array. Each element in this array describes the texture and material combination for a subset of faces in the mesh.

As we learned in Chapter Five, we will need vertex normals if we intend to use the DirectX lighting pipeline. While we know how to generate vertex normals manually with the D3DXComputeNormals function (after we have cloned the mesh to include space for normals), it is certainly nice when the X file can provide those normals directly. If a level editor or 3D modeling application exports vertex normal information, then this is often preferred because it gives the artist the ability to edit those normals to achieve custom lighting effects. Storing vertex normals in the X file is easy enough with the standard X File template called MeshNormals:

```
template MeshNormals
{
   < F6F23F43-7686-11cf-8F52-0040333594A3 >
     DWORD nNormals;
     array Vector normals[nNormals];
     DWORD nFaceNormals;
     array MeshFace meshFaces[nFaceNormals];
}
```

The first DWORD (nNormals) describes the number of normals in the normals array (an array of Vectors) that will follow. There are typically as many vertex normals in this list as there are vertices in the mesh, but sometimes a mesh may choose to save space by assigning the same normal to multiple vertices. If this is not the case, we can just read the normals in and assume that there is a one-to-one mapping of normals to vertices.

If the normal count is different from the number of vertices in the parent mesh vertex list, then this means that the normals are mapped to the vertices in the mesh using indices, similar to the face list. When this is the case, the bottom two members of this template are used. The DWORD nFaceNormals will equal the number of faces in the mesh, and thus the MeshFace array. Each MeshFace object in the array holds indices into the normal list for each vertex in the face. For example, if the first MeshFace contains the indices 10,20,30, then the first vertex of the face uses normal [10] in the normal list, the second vertex uses normal [20] and the third vertex uses normal [30]. More often than not, meshes contain per-vertex normals rather than the MeshFace index scheme and the last two members of this template are not used.

The next X file example adds vertex normals as a new child object for the Mesh.

X File Sample 6: Mesh with Vertex Normals (List Version)

```
Xof 0303txt 0032

mesh{
 6;              // number of vertices
```

```
-5.0;-5.0;0.0;,   // vertex list
-5.0;5.0;0.0;,
 5.0;5.0;0.0;,
 5.0;-5.0;0.0;,
-5.0;5.0;-5.0;,
-5.0;5.0;5.0;;
 2;          // number of faces
 3;0,1,2;,   // face list
 3;3,4,5;;

 MeshMaterialList {// child data object of Mesh
  2; //Two materials in the list
  2; //Two faces in the following material face list

      // material face list
  0,  // the first face uses Material 0
  1;; // the second face uses Material 1

  Material {            // Material 0,child data object of MaterialList
  1.000000;1.000000;1.000000;1.000000;;
  0.000000;
  0.000000;0.000000;0.000000;;
  0.000000;0.000000;0.000000;;
  TextureFilename {"example.bmp";} // child data object of Material
  }

  Material {           // Material 1, child data object of MaterialList
  1.000000;0.000000;0.000000;1.000000;;
  0.000000;
  0.000000;0.000000;0.000000;;
  0.000000;0.000000;0.000000;;
  }
 }

 MeshNormals {               // child data object of Mesh object
 6;                          // one for each vertex
 0.000000;1.000000;0.000000;, // Normal 1
 0.000000;1.000000;0.000000;, // Normal 2
 0.000000;1.000000;0.000000;, // Normal 3....
 0.000000;1.000000;0.000000;,
 0.287348;0.957826;0.000000;,
 0.287348;0.957826;0.000000;;
 2;               // number of faces
 3; 0,2,1;,       // indices of how the normals are mapped to each face
 3; 3,4,5;;
 }

}// end of mesh object
```

Notice that the MeshFace type contains the number of indices and then the indices themselves. In this particular example, the MeshFace list is not necessary since the number of vertices equals the number of normals, but we added the mapping list just to demonstrate the concept. Notice also that the MeshNormals data object is embedded as a child of the mesh object. This makes sense as this object contains the vertex normal information for that mesh.

If you take a look at the 'Mesh Normals' object above, you will see that the first value is 6, which indicates that there are six normals in the list that follows. This simple mesh only has two faces comprised of six vertices in total. After this normal count there follows a list of the six normals. Typically, this would be all the information stored as we have six vertices and six vertex normals and that is all we need to know. For completeness however, in this particular example we demonstrate how there may not be a one-to-one mapping between normals in the normal list and vertices in the vertex list. Following the normal list you can see a value of 2, which means that following this value will be an array of 'Mesh Face' objects used to describe how the normals in the normal list map to vertices in the mesh vertex list. We can see that the first face in this list has 3 indices (which happen to map to the first face in the mesh) and has indices 0, 2 and 1. This means the first vertex of the first face in the mesh should use the first normal in the normal list (normal[0]), the second vertex of the first face of the mesh should use normal[2] from the normal list and the third vertex in the first face of the mesh should use normal[1]. We can also see that the second 'Mesh Face' object has three indices and that the first, second and third vertices of this face use the fourth, fifth and sixth normals in the normal list respectively.

While we have seen how to pair a texture and a material together and assign the combination to faces in the mesh, there is currently no information in our X File example describing how the texture of a given material data object is mapped to a face. Thus, we need a way to store per vertex texture coordinates for the mesh. We do this by embedding another child object, called MeshTextureCoords, inside the mesh object. This standard template is shown below:

```
template MeshTextureCoords
{
  < F6F23F40-7686-11cf-8F52-0040333594A3 >
    DWORD nTextureCoords;
    array coords2d textureCoords[nTextureCoords] ;
}
```

The DWORD describes the number of texture coordinates in the following array of coord2d objects. Typically there will be a set of texture coordinates for each vertex in the mesh vertex list (provided the mesh is textured) and this is always a 1:1 mapping. For example, the first texture coordinate in the array is the texture coordinate set for the first vertex in the mesh vertex list, and so on. While there may be fewer texture coordinates in this list than there are vertices in the mesh, this will still be a 1:1 mapping. If a mesh has 50 vertices, but only the first 30 belong to textured faces and needed texture coordinates, only 30 texture coordinate sets might exist in this array, but they will map to the first 30 vertices in the parent mesh vertex list.

Each pair of texture coordinates in the MeshTextureCoords array are represented as the standard template Coords2d, shown below:

```
template Coords2d
{
   < F6F23F44-7686-11cf-8F52-0040333594A3 >
     float u;
     float v;
}
```

Finally, let's take a look at our X file example with texture coordinates added.

X File Sample 7: Mesh with Texture Coordinates

```
Xof 0303txt 0032

mesh{
 6;               // number of vertices
-5.0;-5.0;0.0;,   // vertex list
-5.0;5.0;0.0;,
 5.0;5.0;0.0;,
 5.0;-5.0;0.0;,
-5.0;5.0;-5.0;,
-5.0;5.0;5.0;;
 2;          // number of faces
 3;0,1,2;,   // face list
 3;3,4,5;;

 MeshMaterialList {// child data object of Mesh
  2; //Two materials in the list
  2; //Two faces in the following material face list

      // material face list
  0,  // the first face uses Material 0
  1;; // the second face uses Material 1

  Material {            // Material 0,child data object of MaterialList
  1.000000;1.000000;1.000000;1.000000;;
  0.000000;
  0.000000;0.000000;0.000000;;
  0.000000;0.000000;0.000000;;
  TextureFilename {"example.bmp";} // child data object of Material
  }

  Material {            // Material 1, child data object of MaterialList
  1.000000;0.000000;0.000000;1.000000;;
  0.000000;
  0.000000;0.000000;0.000000;;
  0.000000;0.000000;0.000000;;
  }
 }

 MeshNormals {               // child data object of Mesh object
 6;                          // one for each vertex
 0.000000;1.000000;0.000000;, // Normal 1
 0.000000;1.000000;0.000000;, // Normal 2
 0.000000;1.000000;0.000000;, // Normal 3....
 0.000000;1.000000;0.000000;,
 0.287348;0.957826;0.000000;,
 0.287348;0.957826;0.000000;;
 2;               // number of faces
 3;0,2,1;,        // indices of how the normals are mapped to each face
 3;3,4,5;;
 }
```

```
 MeshTextureCoords {// child data object of mesh object
 6;                 // number of Texture coordinates one for each vertex

 0.308594;0.152344;,    // texture coordinate 1
 0.007813;0.152344;,    // texture coordinate 2 ....
 0.007813;0.007813;,
 0.308594;0.007813;,
 0.328125;0.156250;,
 0.415681;0.156250;;
 }

}// end of mesh object
```

> **NOTE:** You may be wondering how to assign multiple textures (or materials) to a single face. Unfortunately, this is not possible using the standard templates. That is, multi-texturing is not supported as part of the X file standard, so custom templates would be required to remedy this situation.

There are still many standard templates we have yet to discuss. Some will be discussed later in the chapter, others in the next chapter. But for now, we have a good overall idea of how the X file format works. For a complete list of templates, please be sure to check the X file reference in the SDK documentation. The information we have just discussed will be very useful in the future, especially if you wish to read or write your own X file custom data objects. We do not, however, need to be fully versed in the X file format or its various templates to be able load a scene hierarchy. We demonstrated in Chapter 8 that we did not need to be aware of the way a mesh was laid out in an X file in order to load the mesh using the D3DXLoadMeshFromX function. Nevertheless, this X file discussion has set the stage for our return to frame hierarchies, which we will revisit in the next section in terms of their X file representation.

9.3.3 Hierarchical X Files and Frames

Our focus in this chapter will continue to be on scenes or objects that consist of multiple meshes. When this is the case, the X file will be stored as a frame hierarchy. As already discussed, the hierarchy is constructed using Frames, where a frame takes the form of a tree node in terms of implementation. From the perspective of an X file, these frames will include a transformation matrix stored as a child data object as well as any number of child mesh objects and/or child frame objects. When an X file contains many meshes, they will not generally be defined as top level objects. They are instead stored as frame child objects. It is typical for the X file to contain a single top level Frame (the root frame) and have all other meshes and frames defined either directly or indirectly as children of this root frame. The standard templates for defining hierarchies are Frame and FrameTransformMatrix.

```
template Frame
{
  < 3D82AB46-62DA-11cf-AB39-0020AF71E433 >
  […]
}
```

Frame is an open template that will usually have a FrameTransformMatrix, one or more Mesh objects, and/or other Frames as children. The D3DX loading functions D3DXLoadMeshHierarchyFromX and

the simpler D3DXLoadMeshFromX understand Mesh, FrameTransformMatrix, and Frame template instances as child objects when loading a Frame. These are usually the only objects stored in a frame.

```
template FrameTransformMatrix
{
   < F6F23F41-7686-11cf-8F52-0040333594A3 >
    Matrix4x4 frameMatrix;
}
```

FrameTransformMatrix contains a single member, a Matrix4x4 object, which is another standard template. It is an array of 16 floats describing each element of a 4x4 matrix in row major order:

```
template Matrix4x4
{
   < F6F23F45-7686-11cf-8F52-0040333594A3 >
   array float matrix[16];
}
```

With this information in mind, we now have a complete picture of the form a hierarchy takes inside an X file. The following example will use the automobile hierarchy discussed earlier, but we will leave out the bulk of the mesh data because we already know what mesh data in an X file looks like.. For now, just assume that the mesh data would be contained within the curly braces. Fig 9.4 is a reminder of the automobile hierarchy introduced earlier.

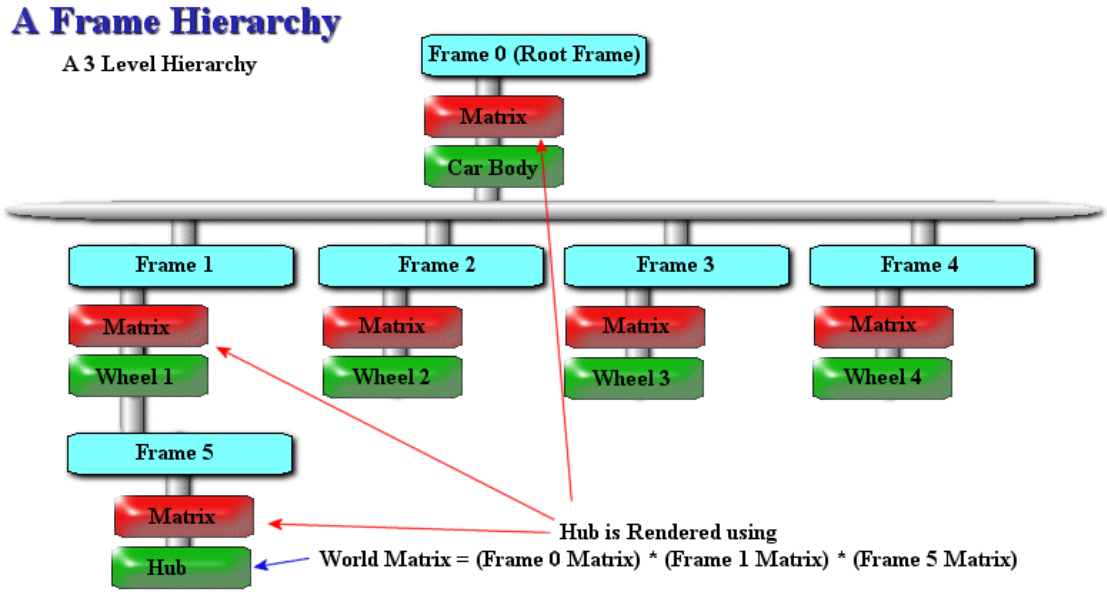


Figure 9.4

The wheel frames will be offset from the root frame at distances of 10 units along the X and Z axes. The root frame has a matrix which sets the object at position (50, 0, 50). Usually, the root frame matrix will either not exist or will be an identity matrix, allowing us to place the entire hierarchy where we want in the world by setting the world matrix and then rendering. However, in this example we will use a root

frame that is offset from the world space origin so that we can see the various matrices being stored in the X file.

Matrix Translation Vectors (Root Matrix now contains position 50,0,50)

Root Matrix	=	**50 , 0 , 50**	(The World Matrix of the Hierarchy)
Wheel 1 Matrix	=	**-10 , 0 , 10**	(RootMatrix * Wheel1Matrix = 40 , 0 , 60)
Wheel 2 Matrix	=	**10 , 0 , 10**	(RootMatrix * Wheel2Matrix = 60 , 0 , 60)
Wheel 3 Matrix	=	**-10 , 0 , -10**	(RootMatrix * Wheel3Matrix = 40 , 0 , 40)
Wheel 4 Matrix	=	**10 , 0 , -10**	(RootMatrix * Wheel4Matrix = 60 , 0 , 40)
Hub Matrix	=	**-5 , 0 , 10**	(Root Matrix * Wheel1Matrix * Hub Matrix = 35,0,10)

And finally, here is the X file that represents everything that we have learned thus far. You should be able to see how the X file is laid out exactly as described by the hierarchy diagram and how the matrices contained in the X file hierarchy are not world transform matrices but are instead relative matrices. Recall that they describe the position of a frame using the parent frame of reference.

X File Sample 8: Hierarchical X File

```
Xof 0303txt 0032

Frame Root
{
  FrameTransformMatrix
  {
    1.000000, 0.000000,  0.000000, 0.000000,
    0.000000, 1.000000,  0.000000, 0.000000,
    0.000000, 0.000000,  1.000000, 0.000000,
   50.000000, 0.000000, 50.000000, 1.000000;;
  }

  Mesh CarBody { mesh data would go in here }

  Frame Wheel1                // Child frame of Root
  {
    FrameTransformMatrix
    {
     1.000000,  0.000000, 0.000000,  0.000000,
     0.000000,  1.000000, 0.000000,  0.000000,
     0.000000,  0.000000, 1.000000,  0.000000,
    -10.000000, 0.000000, 10.000000, 1.000000;;
    }

    Mesh Wheel1 { mesh data would go in here }

    Frame Hub                 // Child frame of Wheel 1
    {
      FrameTransformMatrix
      {
       1.000000, 0.000000, 0.000000, 0.000000,
       0.000000, 1.000000, 0.000000, 0.000000,
       0.000000, 0.000000, 1.000000, 0.000000,
      -5.000000, 0.000000, 0.000000, 1.000000;;
```

```
      }

      Mesh Hub { Hub mesh data goes here }

    } // end child frame Hub

  } // end child frame Wheel 1

  Frame Wheel2                // Child frame of Root
  {
    FrameTransformMatrix
    {
     1.000000, 0.000000, 0.000000,   0.000000,
     0.000000, 1.000000, 0.000000,   0.000000,
     0.000000, 0.000000, 1.000000,   0.000000,
     10.000000, 0.000000, 10.000000, 1.000000;;
   }

   Mesh Wheel2 { mesh data would go in here}

  } // end child Wheel 2

  Frame Wheel3                      // Child frame of Root
  {
    FrameTransformMatrix
    {
     1.000000, 0.000000,   0.000000, 0.000000,
     0.000000, 1.000000,   0.000000, 0.000000,
     0.000000, 0.000000,   1.000000, 0.000000,
    -10.000000, 0.000000, -10.000000, 1.000000;;
    }

    Mesh Wheel3 { mesh data would go in here}

  } // end child frame wheel 3

  Frame Wheel4                      // Child frame of Root
  {
    FrameTransformMatrix
    {
     1.000000,  0.000000,   0.000000,  0.000000,
     0.000000,  1.000000,   0.000000,  0.000000,
     0.000000,  0.000000,   1.000000,  0.000000,
     10.000000, 0.000000, -10.000000,  1.000000;;
    }

    Mesh Whee4 { mesh data would go in here}

  }// end child frame wheel 4

} // end root frame
```

We now have a very solid understanding of how a hierarchy is stored inside an X file. What we will find is that this information is going to be very helpful to us when working with the frame hierarchy after it is loaded. This will be the subject of the sections that follow.

9.3.4 D3DXLoadMeshFromX Revisited

In Chapter Eight we learned how to load single meshes from an X file without making any effort to understand the internals of the file format. However, at the outset of this chapter we stated our concern about the limited nature of the D3DXLoadMeshFromX function. While D3DXLoadMeshFromX will collapse all of the data into a single mesh, it must of course account for the fact that those meshes are individually defined in parent-relative space. So it will need to traverse the hierarchy and cumulatively transform all child meshes using their Frame (and parent Frame) transformation matrices, so that the vertices of the mesh are transformed into the shared hierarchy/scene space. This happens before adding that data to the new D3DXMesh object (which is ultimately returned).

Again, this may be quite acceptable for certain applications that do not require those components to be independently manipulated. This might be the case, using our automobile example, for parked cars that are used as detail objects in a scene. But if we do want to treat those meshes independently, another solution is needed.

There are two ways to extract this hierarchy data from an X file, one more complex than the other. The easiest approach is to use the D3DXLoadMeshHierarchyFromX function, and we will look at this function shortly. The more difficult approach is to use the X file COM objects mentioned earlier in the text and parse the file manually. We will briefly review the COM objects first since this approach provides a more 'behind the scenes' look at the process. There will also be times when you will need to use some of the COM objects alongside the D3DXLoadMeshHierarchyFromX function in order for your application to parse custom data objects. Additionally, you may decide to write your own X File parser, perhaps because you do not want to use the D3DX library functions or because you may not have access to them (or perhaps you wish to write an X File loading module that will compile and work with earlier versions of DirectX). We will only briefly examine these X file interfaces: just enough to provide a high level understanding of the process and to get you started should you decide to pursue the concept on your own at a later date. There is plenty of good reference material in the 'X File Reference' section in the SDK documentation as well, so be sure to consult that documentation as needed.

9.3.5 Loading Hierarchies Manually

DirectX provides several COM objects and interfaces to deal with the manual reading and writing of X files. In order to use these objects and interfaces, we need to include three header files that ship with the SDK. They are contained in the 'Include' directory of the DirectX 9 SDK installation:

1. **dxfile.h** This file contains all the COM interfaces that we need to use to manually process X files.
2. **rmxftmpl.h** This file contains the template definitions for all of the standard X file templates that we have looked at previously, such as 'Frame', 'Mesh' and 'MeshMaterialList' for example.
3. **rmxfguid.h** This file contains all the GUID aliases for the templates so that we can identify object types when processing X file data objects.

NOTE: The X file COM objects are not part of the core DirectX and D3DX libraries. You must make sure that if you intend to use the X File interfaces that you also link the 'd3dxof.lib' file into your project. It can be found in the 'Lib' directory of your DirectX 9 SDK installation directory.

9.3.5.1 The IDirectXFile Interface

The 'd3dxof.lib' library exposes a single global function call to create our initial interface. This interface will serve as the gateway to the rest of the X file interfaces from within our application. This is similar to our use of Direct3DCreate9 to create our initial IDirect3D9 interface, which then serves as a means to create an IDirect3DDevice9. We call the DirectXFileCreate global function with the address of an IDirectXFile interface pointer. On function return, this will point to a valid DirectXFile object.

```
IDirectXFile * pDXFile = NULL;
DirectXFileCreate ( &pDXFile );
```

This simple interface contains three methods. One of those methods handles saving data to an X file and we will not concern ourselves with this for the time being. For now we will focus on the other two methods which are used in the file loading process.

HRESULT RegisterTemplates(LPVOID *pvData*, DWORD *cbSize*);

This method is the primary means by which we inform DirectX of the templates that we wish to parse in the X file. Each template must be registered with the loading system, or else data objects of that type inside the X File will not be recognized later during parsing. The first parameter points to a buffer that contains the template definitions we wish to register and the second parameter indicates the size (in bytes) of this buffer.

All of the standard templates that we looked at earlier (as well as many others) are already contained in a buffer called `D3DRM_XTEMPLATES` that is defined in 'rmxftmpl.h'. This is why we included this header file. The same header file defines the size of this buffer (3278 bytes) and assigns this value to `D3DRM_XTEMPLATE_BYTES`. To register the all of the standard X file templates, we would use code like this:

```
IDirectXFile * pDXFile = NULL;
DirectXFileCreate ( &pDXFile );
pDXFile->RegisterTemplates( (VOID*) D3DRM_XTEMPLATES, D3DRM_XTEMPLATE_BYTES );
```

At this point, if we were not using custom templates we could begin parsing immediately. If the X file does contain custom data objects (i.e. data objects which are not part of the X file standard), then their templates will also need to be registered. We can build the template string manually if we wish and pass it into the function as shown next:

```
char *szTemplates = "xof 0303txt 0032\
               template Contact { \
                  <2b934580-9e9a-11cf-ab39-0020af71e433> \
                  STRING Name ;\
```

```
                  STRING Address1;\
                  STRING Address2;\
                  STRING PostCode;\
                  STRING Country;\
               } \
            template Hobbies  { \
                  <2b934581-9e9a-11cf-ab39-0020af71e433> \
                   DWORD nItems;\
                   array STRING HobbyNames;\
            }";

IDirectXFile * pDXFile = NULL;
DirectXFileCreate( &pDXFile );
pDXFile->RegisterTemplates( (void *) szTemplates , strlen ( szTemplates ) );
```

In this example we have registered two custom templates, called ‘Contact’ and ‘Hobbies’, which could be used to store personal information about a person. Once these templates have been registered, if data objects of either type are found in the X file, the loading system will process them as valid objects.

The RegisterTemplates function will search the passed buffer for valid templates and register them with the loading system. Recall from our earlier discussion that it is common practice to define templates at the top of the X file itself, before the actual data object definitions. If this is the case, we could instead load the entire X file into a memory buffer (just performing a block read) and then pass this buffer (containing all X file data) into the RegisterTemplates function. The function will find the templates listed at the top of the file and will automatically register them. This means you will not have to hardcode template definitions into your project code and that you can easily change the template definitions just by altering the templates in the X file. They will automatically be registered each time the application is run. If the X file contained template definitions for all the standard templates that it uses, then registering the standard templates as a separate step would also be unnecessary.

After we have registered any standard and/or custom templates we intend to use, it is time to call the second function of the IDirectXFile interface. This function, IDirectXFile::CreateEnumObject, opens the X file and creates a top level data enumerator object which is returned as a pointer to an IDirectXFileEnumObject interface. We can use this interface to step through each top level data object in the X file one at a time and extract their data.

```
HRESULT CreateEnumObject( LPVOID pvSource,
                          DXFILELOADOPTIONS dwLoadOptions,
                          LPDIRECTXFILEENUMOBJECT *ppEnumObj );
```

The first parameter is a pointer to a buffer whose meaning depends on whether we are loading the X file from disk, resource, or memory buffer. This is specified by the second parameter which should be one of the DXFILELOADOPTIONS flags:

DXFILELOAD_FROMFILE - If this flag is used for the second parameter, then the first parameter to the CreateEnumObject function is a string containing the file name of the X file.

DXFILELOAD_FROMRESOURCE - If this flag is used as the second parameter, the first parameter should be a DXFILELOADRESOURCE structure. For more details on this structure, see the SDK documentation as we will not be covering it here.

DXFILELOAD_FROMMEMORY - If this flag is used as the second parameter, the first parameter should be a DXFILELOADMEMORY structure. For more details on this structure, check out the SDK documentation as we will not be using it in this chapter.

The complete process for registering the standard templates and then creating our enumeration object to process X file data would be as follows:

```
IDirectXFile            *pDXFile     = NULL;
IDirectXFileEnumObject  *pEnumObject = NULL;
DirectXFileCreate ( &pDXFile );
pDXFile->RegisterTemplates( (VOID*) D3DRM_XTEMPLATES ,
                                    D3DRM_XTEMPLATE_BYTES );

pDXFile->CreateEnumObject( "MyCube.x" , DXFILELOAD_FROMFILE, &pEnumObject );
```

> **Note:** This section is intended only as a brief introduction to the X file interfaces for those of you interested in parsing your X files manually. In this chapter, our demos will not be using these interfaces, and will instead be doing things the easy way using the D3DXLoadMeshHierarchyFromX function. This information is still useful though as later on we will discuss how we can use the D3DXLoadMeshHierarchyFromX function to parse custom X file templates. In that case, a thorough understanding of the X file format and the IDirectXFileData interface is necessary.

After the code shown above has been executed, the application has a pointer to an IDirectXFileEnumObject interface. This interface exposes methods which the application can use to start processing top level data objects stored in the X file.

9.3.5.2 The IDirectXFileEnumObject Interface

Once we have an IDirectXFileEnumObject interface, we can call its GetNextDataObject method to retrieve each top level data object stored in the X file, one at a time. Typically this will happen in a loop, where the data objects (returned to the application as DirectXFileData objects) are processed until no more top level data objects exist in the file.

For each top level data object that exists in the file (ex. Mesh, Frame, etc.), the data is returned to the application as an IDirectXFileData interface. The application can use the methods of this interface to extract the data as needed. Note that this function only returns top level objects, not child objects like, for example, a MeshMaterialList object which would be embedded as a child object inside a Mesh data object. Because all top level objects must be data objects, and not data reference objects, the enumerator can safely make this assumption.

The following code shows how we might set up a loop to process each top level object in the file.

```
LPDIRECTXFILE           pDXFile;
LPDIRECTXFILEENUMOBJECT pEnumObject;
LPDIRECTXFILEDATA       pFileData;
const GUID *pGuid;

DirectXFileCreate (&pDXFile);
pDXFile->RegisterTemplates( (VOID*) D3DRM_XTEMPLATES , D3DRM_XTEMPLATE_BYTES);
pDXFile->CreateEnumObject( "MyXFile.x" , DXFILELOAD_FROMFILE, & pEnumObject));

while ( SUCCEEDED ( pEnumObject->GetNextDataObject(&pFileData)))
{
     pFileData->GetType(&pGuid);
     if (*pGuid==TID_D3DRMMesh)     GetMeshData  ( pFileData );
     if (*pGuid==TID_D3DRMFrame)    GetFrameData ( pFileData );
     pFileData->Release();
}

pEnumObject->Release();
pDXFile->Release();
```

The code shows that once we have created the top level enumerator object, we repeatedly call its GetNextDataObject method to access the next top level data object in the X file. The actual data contained within the top level object is contained inside the returned IDirectXFileData interface.. Once we have an IDirectXFileData interface, we call its GetType function to test its GUID (i.e. the GUID of its template) against the ones we wish our application to support. In this example you can see that we are testing for top level Mesh and Frame objects only. The aliases for their GUIDs (and all of the standard template GUIDs) are listed in the 'rmxfguid.h' header file. TID_ is an abbreviation for Template ID.

Once we have identified a supported type in the above example code, we call some application defined functions to process the data stored in the returned IDirectXFileData object. In this example it is assumed that the GetMeshData and GetFrameData functions are application defined functions which will use the passed IDirectXFileData interface to extract the data into some meaningful structures (an ID3DXMesh for example). We will study the IDirectXFileData interface in a moment when we learn how to extract the data from a loaded data object.

There are three similar methods in the IDirectXFileEnumObject interface that we can use to access top level data objects defined in the X file. The first is called GetNextDataObject and is the one we have used in the above code example. All three data object retrieval methods of the enumerator interface are listed next:

```
HRESULT GetNextDataObject(LPDIRECTXFILEDATA *ppData);
HRESULT GetDataObjectByName(LPCSTR szName,LPDIRECTXFILEDATA *ppData);
HRESULT GetDataObjectById(REFGUID rguid, LPDIRECTXFILEDATA *ppData);
```

The **GetNextDataObject** function is called to fetch the next top level data object from the file. We saw this being used in the code example shown previously. The function is passed the address of an IDirectXFileData interface pointer which on function return will point to a valid interface which can be

used to extract the data from the data object. If the X file contains six top level data objects, this function would be called six times, each time fetching the next data object until it fails.

The **GetDataObjectByName** function lets us search the X file for an object with the name passed in. The second parameter is the address of an IDirectXFileData interface pointer which, if a top level object with a matching name exists in the file, will be used to extract data from the requested object. The example data object shown below, which is of type 'TestTemplate', would have the name 'MyFirstValues'. Fetching data objects by name is very useful if you know the name of a specific object you wish to load. This allows you to load this data object without having to step through all other top level data objects in the file.

The **GetDataObjectById** works in exactly the same way as the previous function, but searches on data object GUIDs rather than names. Although we have not mentioned it until now, each data object in an X file can also have its own **unique** GUID in addition to the template GUID. For example, let us say that we defined our data for a TestTemplate object as follows:

```
TestTemplate MyFirstValues {
  <78A5640B-1A66-4CD0-B1A4-3E2060429130>
  1.000000;
  2.000000;
  3.000000;
}
```

We have now added a GUID to the actual data object itself. The GUID that we included in the object above (remember this is not a template, but an actual instance of a TestTemplate object) can be used to identify this specific object in the file. Therefore, the template GUID can be used to identify all data objects of the same type in the X file and the data object GUID can be used to identify a single data object. This allows an application to find it easily even if the X file contains many data objects based on the same template.

9.3.5.3 The IDirectXFileData Interface

The IDirectXFileData interface encapsulates the reading and writing of X file data objects. This is true whether we are using the COM interfaces to manually parse the X file as discussed in the last section, or whether we use D3DXLoadMeshHierarchyFromX and custom data blocks have to be processed. Whether we intend to use the X File COM objects for loading X files manually, or intend to use the much easier D3DXLoadMeshHierarchyFromX function, exposure to this interface cannot be avoided. We will see shortly that when the D3DXLoadMeshHierarchyFromX function encounters a custom data object (for which D3DX has no knowledge of how to load), it will use a callback mechanism to send this data object to the application (as an IDirectXFileData). The application can then extract the data contained using the methods of this interface.

We saw earlier that when we use the IDirectXFileEnumObject interface to retrieve top level data objects in an X file, it actually returns this interface for each data object found. This interface encapsulates the data contained in the data object, such as the data for a mesh, frame or material object.

The IDirectXFileData interface exposes eight methods. Three of them are used primarily in the building and saving of X files and five are used for reading and processing the data stored in the X file data object. Since we are discussing loading X files at the moment, we will concentrate on these five methods for now. They are listed next with short descriptions. To make these functions easier to explain, we will use an example template and an object of that template type in our discussions:

```
template TestTemplate
{
  <A96F6E5D-04C4-49C8-8113-B5485E04A866>
  FLOAT TestFloat1;
  FLOAT TestFloat2;
  FLOAT TestFloat3;
}
```

This is a simple custom template with a GUID (which it must have) describing a data object that has three floats. Let us declare an instance of this template to work with:

```
TestTemplate MyFirstValues
{
   <78A5640B-1A66-4CD0-B1A4-3E2060429130>
  1.000000;
  2.000000;
  3.000000;
}
```

Note that this object also has its own GUID and no other object in the file will use this GUID. The template GUID describes the type of object this is (of which there may be many objects of this type in the file) and the object GUID identifies the exact instance of the template object. While per-object GUIDs are optional, it is a handy feature that you will no doubt see being used from time to time.

`HRESULT GetName( LPSTR pstrNameBuf, LPDWORD pdwBufLen );`

This function retrieves the name of the data object as specified in the X file. So in our example, because we specified "`TestTemplate MyFirstValues`" in the X file, the value returned would be "`MyFirstValues`". We are not required to specify a name when we instantiate an object in an X file; in fact, we saw examples earlier that simply instantiated an object using only a template name without assigning the object its own name. However, it can be handy to be able to reference the data later on elsewhere in the file, so it is good practice. Of course, this is often out of our hands as the X file will usually be created by some 3rd party modelling program for which we have no control over its X file export process. In the case of data reference objects, they naturally use per-object names so that they can be referenced by other data objects elsewhere in the file. Let us take a look at its two parameters.

LPTSTR pstrNameBuf
This first parameter is a pointer to a pre-initialised char buffer that should contain the resulting name on function return. It can be as large or small as you require. The name will be copied into this buffer.

LPDWORD pdwBufLen
This is a pointer to a `DWORD` which is used as both an 'In' and 'Out' parameter. The value, contained in the `DWORD` (that you pass in), must specify the maximum length of the string buffer that was passed in to

the previous parameter. When the function returns, this `DWORD` will be filled with the *actual* buffer length required to store the string, including the terminating character. If the original value you passed in specifies a value that is not large enough to return the entire string, then this function will return `D3DXFILEERR_BADVALUE`.

Passing NULL for the `pstrNameBuf` parameter will fill out the DWORD with the length of the buffer required to store the returned string (the complete name of the object). Note that the same rules still apply: the input value you pass in here must still be large enough (even if you do pass NULL as the string parameter). In both cases, an error will be returned if this is not the case. This allows you to test whether a certain buffer length will be large enough to store the returned string.

`HRESULT GetType (const GUID **ppguid);`

One of the first things we want to know about an IDirectXFileData interface is what type of data object it represents (Mesh, Frame, Material, etc.) so that we can select the appropriate processing path or determine if our application even wants to process it at all. GetType allows us to retrieve a pointer to the GUID of the template type for the data object. In our example, the GetType method would return the GUID of the TestTemplate type. We used this function earlier when enumerating through the file:

```
while ( SUCCEEDED ( pEnumObject->GetNextDataObject(&pFileData)))
{
     pFileData->GetType(&pGuid);
     if (*pGuid==TID_D3DRMMesh)  GetMeshData  ( pFileData );
     if (*pGuid==TID_D3DRMFrame) GetFrameData ( pFileData );
     pFileData->Release();
}
```

We pass in the address of a GUID pointer to retrieve a pointer to the GUID object itself. As you can see, this is a const pointer so we are not allowed to modify the GUID.

`HRESULT GetId( LPGUID pGuid );`

This method will retrieve the GUID for the object being referenced. Unlike the type GUID specified in a template declaration, this is the per-object GUID discussed previously. In our example object ('MyFirstValues'), we have included a GUID, so this is the value that will be retrieved here. We discussed earlier how to use the 'DEFINE_GUID' macro so that you can build aliases (such as TID_D3DRMMesh) for your own GUIDs and use them for type comparison.

LPGUID pGuid
This method takes a single parameter. A pointer to a GUID structure to be filled with the GUID data on function return. If no GUID exists in the data object, it will be filled with zeros.

`HRESULT GetData( LPCSTR szMember, DWORD *pcbSize, void **ppvData ) ;`

This is the primary method for retrieving the object data that was loaded from the file. If the data object represented by this interface was a Mesh object for example, the GetData method allows us to access the underlying vertex and index data stored within. It accepts three parameters which are described below.

LPCTSTR szMember

For this parameter, there are two options. The first option is to pass in the name of a member variable in the data object that you wish to retrieve the value for (ex. "`TestFloat1`" in our current example). This allows us to retrieve the variables of a data object one at a time, or get the values only for specific variables that we may be interested in. The second option is to pass NULL, which indicates that all of the data for the entire data object should be retrieved.

DWORD *pcbSize

On function return, this `DWORD` pointer will contain the total size of all of the data retrieved.

void **ppvData

As the third parameter, the application should pass the address of a void pointer. On function return this will point to the start of the data buffer we requested. Whether this buffer contains just a single member variable of the object or the entire data set itself depends on whether or not NULL was passed as the first parameter. The application never gets to own this data as it is issued a pointer to the API's own internal data. For this reason, we must copy the information pointed at into memory that our application does own so that we can modify it and rely on its persistence.

If we request a pointer for the entire data buffer, then the template itself should be our guide for determining the offsets for where each variable is stored in that buffer. Look again at our example template:

```
template TestTemplate
{
  <A96F6E5D-04C4-49C8-8113-B5485E04A866>
  FLOAT TestFloat1;
  FLOAT TestFloat2;
  FLOAT TestFloat3;
}
```

We can see that the buffer will contain three floats. Note that the GUID is *not* part of the data buffer, as there are separate methods for accessing the GUID as previously discussed. Using the above example, we know that the pointer would initially point to the float value 'TestFloat1'. Assuming 32-bit floating point precision is being used, we know that incrementing a BYTE pointer by four bytes would now point it at the value assigned to 'TestFloat2', and so on. Of course, the most common and easiest ways to access the data would be to either use a float pointer to step though the values with a simple pointer increment (for this example) or to build a three float structure that mirrors the data area and do a block copy. The returned void pointer could then be cast to a pointer of this type and the members accessed/copied intuitively. We might imagine we have defined a 3 float structure called 'MyXStruct'. We could then cast the returned void pointer and extract the data into three application owned variables (called foo1, foo2 and foo3) as shown below. vpPointer is assumed to be the void pointer to the data buffer returned by the GetData method:

```
MyXStruct *data = vpPointer;
foo1 = MyXStruct->TestFloat1;
foo2 = MyXStruct->TestFloat2;
foo3 = MyXStruct->TestFloat3;
```

This method is inadvisable however because the compiler may insert extra padding into your structures to make the members 32-bit aligned for increased performance. As this invisible padding is not inserted into the retrieved data object buffer, a mismatch can occur between the offsets of each member in the structure and their corresponding offsets in the data buffer. We will have more discussion about this problem a bit later.

The previous four methods of the IDirectXFileData interface that we have discussed demonstrate how to identify a data object and extract its data. We have discovered however, that a data object in an X file, like Meshes for example, may also contain child data objects. Thus, the IDirectXFileData interface may point to a data object that has child objects. We need a way to not only to process its data as previously discussed, but also a way to perform the same processing on any child objects it may contain. This is definitely necessary as the IDirectXFileEnumObject interface (which we have used for data object retrieval thus far), only retrieves top level data objects. This would leave us no way to access any embedded child objects should they exist.

The IDirectXFileData interface has a method that works very much like the IDirectXFileEnumObject::GetNextDataObject function. It allows us to step through its child objects one at a time retrieving IDirectXFileData interfaces for each of its children. This final method (called GetNextObject) provides the ability for our application to reach any object in a complete hierarchy of X file data objects. Not surprisingly, it is common for the loading of an X file to be done as a recursive process.

```
HRESULT GetNextObject( LPDIRECTXFILEOBJECT *ppChildObj );
```

The GetNextObject method allows us to iterate through all the child objects just as we did earlier for top level objects. We will continue to call this method until it returns `DXFILEERR_NOMOREOBJECTS`. Essentially, the interface maintains an internal ‘current object’ pointer so that it knows how far through the list it is. Note that this is a one-time only forward iteration process; once we have traversed the list, we cannot restart from the beginning.

This function has a single parameter:

LPDIRECTXFILEOBJECT *ppChildObj
This pointer will store the address of the current child to be processed. We might expect that the child data object would be returned via an IDirectXFileData interface (as with top level objects) but we are actually returned an IDirectXFileObject interface instead. This is because, unlike top level data objects which are always explicit data objects, child data objects may be specified as either actual data objects or data *reference* objects. You will recall from our earlier X file examples that we showed a top level material object being referenced using a child data reference object in a Mesh. The `IDirectXFileObject` interface is actually the base class for three derived classes: `IDirectXFileData` (which we are now familiar with), `IDirectXFileDataReference`, and `IDirectXFileBinary`. We are not told what type of derived interface this object is via this function, so we must `QueryInterface` the base pointer to find out. The GUIDs for the three types of derived interfaces that may be returned by this function are:

```
IID_IDirectXFileData
IID_IDirectXFileDataReference
IID_IDirectXFileBinary
```

If the IDirectXFileData::GetNextObject method returns a data reference object, we can determine this by calling QueryInterface on the base pointer and passing in IID_IDirectXFileDataReference as the interface GUID. If it is a data reference object, a valid IDirectXFileDataReference interface will be returned for the object. If we do find a child data reference object (instead of a normal child data object), we have to handle it slightly differently. A data reference object does not actually contain data. It is analogous to a pointer in many ways, pointing to a data object elsewhere in the file. We must use the IDirectXFileDataReference interface to resolve this reference and point us at the actual data object which contains the data we need to extract.

We have already discussed the first interface type (IDirectXFileData) and know how to extract the data for a child data object. Let us now look at the other two interfaces which may be returned.

9.3.5.4 The IDirectXFileDataReference Interface

While the top level enumerator will always return an IDirectXFileData interface (remember, top level objects cannot be data reference objects or binary objects) this is not the case for child objects. If the IDirectXFileData::GetNextObject function returns a data reference object, then we know that it does not contain data, but rather acts like a pointer to a data object elsewhere in the file. To access the actual data, we will use the following method:

```
HRESULT Resolve( LPDIRECTXFILEDATA * ppDataObj );
```

This function resolves the reference by returning an `IDirectXFileData` interface to the actual data of the object being referenced. From our application's perspective, from this point on, we are dealing with a normal data object. We can work with the IDirectXFileData interface to extract the data as discussed previously, knowing that it is a child of the outer data object.

There are two other methods exposed by this interface: GetID and GetName. They are identical to the calls of the same name that were discussed when we studied the IDirectXFileData interface (both classes derive them from the same base class).

9.3.5.5 The IDirectXFileBinary Interface

This interface allows us to retrieve any binary data objects that may be stored in the X file, even in cases where the file is a formatted text version. We will not be using this at interface at all, so be sure to study the SDK documentation for more information.

You now have more than enough information to get started on your own X file parser if you care to write one. Again, with DirectX 9, there is probably little need given the utility functions provided. But if this is so, you might be wondering why we just spent so much time examining the X file format in such detail. As it turns out, understanding the X file format is still very useful, even when using the D3DXLoadMeshHierarchyFromX function. This is especially true if we need to parse custom objects. Furthermore, if you wanted to write a level editor or an X file exporter in the future, perhaps as a plug-in or tool for a game that you were developing, you would want to have at least this level of familiarity with X files so that you can write your file data correctly. And not to be overlooked, now that you understand how to construct X files manually, you could build simple frame hierarchies using a text editor like Notepad in the absence of an available scene editor.

> **Note:** While we are not going to look at a complete source code example of loading an X file using the 'X File' interfaces discussed, the DirectX SDK ships with source code to a mesh loading class in CD3DFile.h and CD3DFile.cpp which you can use in your own applications. CD3Dfile.cpp can be found in the 'Samples\C++\Common\Src' folder of your DirectX 9 installation. Here you will see all of the interfaces we have discussed being used to manually load an X file and maintain its hierarchical structure.

9.4 D3DXLoadMeshHierachyFromX

D3DXLoadMeshHierarchyFromX is very similar to the D3DLoadMeshFromX function studied in the last chapter. The key difference is that it supports the loading of multiple meshes and creates an entire frame hierarchy in memory. While this function removes the burden of having to become familiar with the X file format and process them using the COM objects discussed in the last section, we still need to understand the X file interfaces if we wish to process custom objects. We will look at some custom objects later in the course, but for now we will concentrate on loading hierarchies that use only the standard templates.

As you can see in the declaration that follows, there are some new D3DX structures and classes we will need to learn about before we can use this function. Furthermore, we must also learn how to derive a COM interface so that we can provide the function with application-defined memory allocation routines for mesh and frame data.

```
HRESULT D3DXLoadMeshHierarchyFromX
(
    LPCTSTR                     Filename,
    DWORD                       MeshOptions,
    LPDIRECT3DDEVICE9           pDevice,
    LPD3DXALLOCATEHIERARCHY     pAlloc,
    LPD3DXLOADUSERDATA          pUserDataLoader,
    LPD3DXFRAME*                ppFrameHierarchy,
    LPD3DXANIMATIONCONTROLLER*  ppAnimController
);
```

We will discuss the parameter list slightly out of order so that we can have a better idea of how it all works.

LPCTSTR Filename
This is a string containing the name of the X file we wish to load. This X file may contain multiple meshes and/or a frame hierarchy. This function will always create at least one frame (the root frame), even if the X file contains only a single top level mesh and no actual frames. In such a case, the mesh will be an immediate child of the root frame (also returned by this function in the sixth parameter).

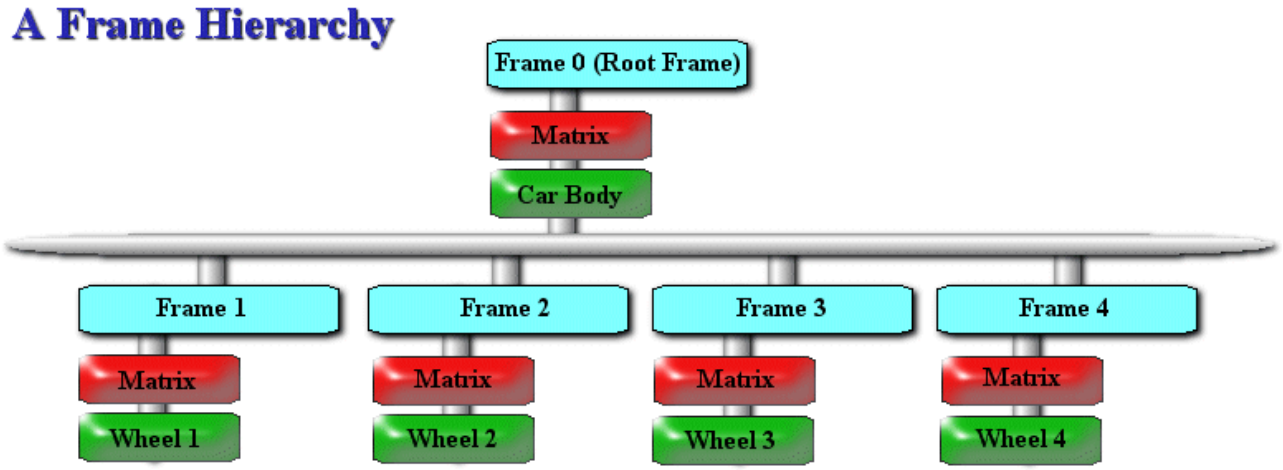
DWORD MeshOptions
These are the standard mesh options that we pass to mesh loading functions. We use a combination of zero or more members of the D3DXMESH enumerated type to specify the memory pools for mesh data, dynamic buffer status, etc.

Note that these flags may be ignored or altered by D3DX, and meshes may be created in a different format. Shortly we will see that during the loading of each mesh, we get a chance to test them (via a callback mechanism) for the desired options. If the mesh format that D3DX has chosen is not what we want, we can clone another mesh and then attach it to the parent frame instead.

LPDIRECT3DDEVICE9 pDevice
This is a pointer to the device that will own the resource memory.

LPD3DXFRAME *ppFrameHeirarchy
The sixth parameter to the D3DXLoadMeshHierachyFromX function is vitally important, since without it we would not be able to access the loaded hierarchy. We pass in the address of a pointer to a D3DXFRAME structure and on function return this will point to the root frame of the hierarchy. We can then use this pointer to access the root frame, its child frames and any child meshes it contains. This will be the doorway to our loaded hierarchy, allowing us to completely traverse the hierarchy to any level we desire.

The D3DXFRAME structure is used by D3DX to store the information for a single frame in the loaded hierarchy. A hierarchy will consist of many of these structures linked together. If you look at the following diagram of a hierarchical X file, followed by the D3DXFRAME structure definition, you will see that the structure itself mirrors the information that the diagram would suggest a frame would need to store.



```
typedef struct _D3DXFRAME
{
    LPTSTR                  Name;
    D3DXMATRIX              TransformationMatrix;
    LPD3DXMESHCONTAINER     pMeshContainer;
    struct _D3DXFRAME *     pFrameSibling;
    struct _D3DXFRAME *     pFrameFirstChild;
} D3DXFRAME, *LPD3DXFRAME;
```

We see that a frame includes a name as its first member. This can certainly be useful if needed to search for a specific component in your hierarchy or if you wanted to use it for display purposes.

The 4x4 D3DXMATRIX member stores the parent-relative transformation we discussed in the early part of the chapter. This is a relative matrix describing the position and orientation of the frame using its parent frame (if one exists) as its frame of reference.

The D3DXMESHCONTAINER structure stores a single set of mesh data (geometry, material, etc.) local to this node. It stores all the data for a single mesh. Note that we will see momentarily that this structure is capable of serving as a node in a linked list of mesh containers (via a pNext pointer) should multiple meshes actually need to be stored in a single frame in the hierarchy. The mesh container pointer in the D3DXFrame structure then is potentially the head of that linked list. It is more typical for a single mesh to be stored in a frame (if any) and therefore this pointer will often point to a single D3DXMESHCONTAINER structure whose next pointer is NULL. We will examine this structure in more detail in a moment.

The last two members are both pointers to other D3DXFRAME structures. The pFrameSibling pointer points to frame structures that are at the same level in the hierarchy as the current frame (i.e., they share the same parent frame). This pointer will be set to NULL for the root frame. If you recall our automobile example, all four of the wheel frames were child frames of the root frame and they are therefore sibling frames. The pFrameFirstChild pointer is essentially the head of a linked list of sibling frames. In the automobile example, the root frame pFrameFirstChild pointer would point at the first wheel frame. That wheel frame would then be linked together with its siblings using the pFrameSibling pointers. So the root frame (Frame 0) will have its pFrameFirstChild pointing at Frame 1. Frame 2 would be pointed to by Frame 1's pFrameSibling pointer. Frame 3 would be linked using Frame 2's pFrameSibling pointer, and so on. Thus the pFrameSibling pointer acts like the generic 'next' pointer in a standard linked list implementation. To traverse the hierarchy from the root down, we start at the pFrameFirstChild after processing the root, step through the sibling linked list, and then process the next set of children in the same fashion, and so on.

So, we have seen that when we use the D3DXLoadMeshHierarchyFromX function, D3DX will construct an entire hierarchy of D3DXFRAME structures in memory. The root frame is returned by the function which can then be used by the application to traverse and render the hierarchy. Wherever a mesh is attached to a frame in the hierarchy, the corresponding D3DXFRAME structure will have a valid pointer to a D3DXMESHCONTAINER structure. This structure contains all of the information for the mesh that D3DX has loaded and attached to the frame on our behalf. It also contains the actual ID3DXMesh interface for the mesh.

The D3DXMESHCONTAINER is defined as follows:

```
typedef struct _D3DXMESHCONTAINER
{
      LPTSTR                      Name;
      D3DXMESHDATA                MeshData;
      LPD3DXMATERIAL              pMaterials;
      LPD3DXEFFECTINSTANCE        pEffects;
      DWORD                       NumMaterials;
      DWORD                       *pAdjacency;
      LPD3DXSKININFO              pSkinInfo;
      struct _D3DXMESHCONTAINER   *pNextMeshContainer;
} D3DXMESHCONTAINER, *LPD3DXMESHCONTAINER;
```

LPTSTR Name

Like a frame in the hierarchy, we can assign a name to the mesh stored in the container. This is almost always the name of the actual mesh (e.g. 'Door Hinge 01') that is assigned to the mesh inside the 3D modeller.

D3DXMESHDATA MeshData

This structure stores the mesh object that D3DX has loaded/created. It contains either a pointer to a regular ID3DXMesh, an ID3DXPMesh, or an ID3DXPatchMesh (which we will not discuss until later in the curriculum). Since the mesh can only be one of these types at any given moment, the pointers can share the same memory by specifying them as a union.

```
typedef struct _D3DXMESHDATA
{
   D3DXMESHDATATYPE     Type;
   union
   {
      LPD3DXMESH        pMesh;
      LPD3DXPMESH       pPMesh;
      LPD3DXPATCHMESH   pPatchMesh;
   } // End Union
} D3DXMESHDATA, *LPD3DXMESHDATA;
```

The Type member allows us to determine which of the three mesh pointers is active. It will be set to one of the following values:

```
D3DXMESHTYPE_MESH
D3DXMESHTYPE_PMESH
D3DXMESHTYPE_PATCHMESH
```

LPD3DXMATERIAL pMaterials

This is an array of `D3DXMATERIAL` structures which store the material and texture data used by the mesh object specified by the previous member. This is identical to the information returned by the D3DXLoadMeshFromX function. Each D3DXMATERIAL structure in this array describes the texture and material for the corresponding subset in the mesh. This array is unique to this mesh container and information is not shared between separate containers in the hierarchy. Therefore, we must make sure when processing this data that we do not load the same texture

multiple times. This is easy enough to address as D3DX will provide our application with the ability to process each mesh container before its gets added to the frame hierarchy. This is accomplished via a callback mechanism that D3DX uses to communicate with the application during the hierarchy construction process.

LPD3DXEFFECTINSTANCE pEffects
Just as the D3DXLoadMeshFromX function returned a buffer of effect instances (one for each subset in the mesh), this array represents the same information for the mesh attached to this mesh container. Effects will be covered in the next module in this series.

DWORD NumMaterials
This variable contains the number of subsets in the mesh attached to this mesh container. This tells us both the number of D3DXMATERIAL structures in the pMaterials array and the number of D3DXEFFECTINSTANCEs in the pEffects array.

DWORD *pAdjacency
This is the standard array describing the face adjacency within the mesh.

LPD3DXSKININFO pSkinInfo
This member stores the skin information for the mesh if the mesh uses the skinning technique for animation. We will cover skinning in detail when we discuss skinning and skeletal animation in Chapter 11. We will ignore this parameter at this time.

struct _D3DXMESHCONTAINER *pNextMeshContainer
The mesh container can act as a node in a linked list. This is a pointer to the next mesh container in the list should multiple meshes be assigned to a single frame in the hierarchy.

So we can see that a mesh container stores what is essentially all of the information that we retrieve in a single call to `D3DXLoadMeshFromX`. In effect, D3DX is doing exactly this, with the exception that instead of collapsing all of the mesh data into a single mesh object, this process is performed for each mesh in the hierarchy. Each mesh's materials, effects, adjacency, and other data is loaded into this container, almost as if we had loaded numerous separate single mesh .X files ourselves into a hierarchy of D3DXFRAME structures.

Fig 9.5 gives us another look at our simple automobile hierarchy, using the structures we have discussed thus far. This diagram demonstrates the hierarchy that would be created by the D3DXLoadMeshHierarchyFromX function if the X file being loaded contained our multi-mesh automobile. This is the version that has a hubcap mesh connected to the first wheel to better show a multi-level hierarchy. In the following example, no two meshes share the same immediate parent frame, but if multiple meshes were immediate children of the same parent frame, the parent frame would point to the first of these mesh containers, and then all the remaining mesh containers of this frame would be linked together (much like sibling frames) using their pNextMeshContainer pointers. Pay special attention to how the frames are linked together. Notice how frames 1-4 are linked by their sibling pointers. This entire row of children is attached to the parent via the root node's child pointer (points to frame 1). Also notice how each frame has a mesh container attached which describes all the information and contains all the data associated with that mesh. The root frame would be returned to the calling

application by the loading function. The application can use this root frame to traverse the hierarchy and access and render its mesh containers.

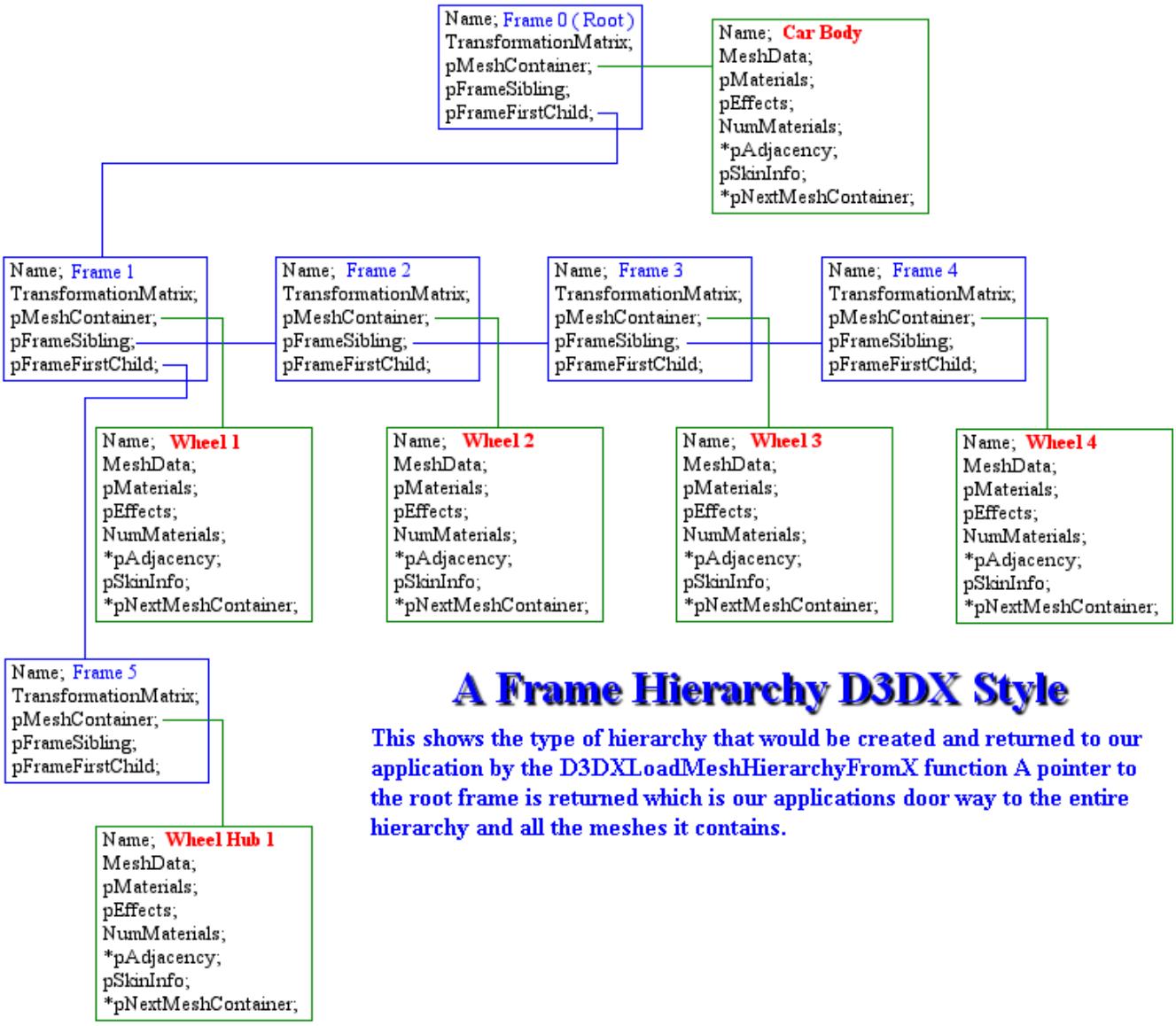


Figure 9.5

While we now know what the hierarchy will look like when the function returns, we still need to finish our examination of the parameters for D3DXLoadMeshHierarchyFromX. As it turns out, hierarchy construction is not completely automated. The application will need to play a role in the allocation of resource data (frames and mesh containers) as the hierarchy is assembled. This is important because we want the application to own the memory used by the hierarchy. If this were not the case, then routine hierarchy alterations or de-allocation of hierarchy components by the application would result in exceptions being generated.

LPD3DXALLOCATEHIERARCHY *pAlloc*

As the fourth parameter to the D3DXLoadMeshHierarchyFromX function, we must pass in a pointer to an application defined class that is derived from the ID3DXAllocateHierarchy interface. It is this parameter which is in many ways the key to the whole loading process. This interface cannot be instantiated on its own because its four member functions are pure virtual functions (they contain no function bodies). We must derive from it, and overload several functions in order to plug our application specific functionality into the D3DX hierarchy loading process.

> **Note:** Some versions of the DirectX 9 SDK suggest that this interface derives from IUnknown (much like the other COM interfaces we have seen in DirectX). This is not the case. It is more correct to think of this interface as being just a pure virtual C++ base class.

This interface is declared using all the standard interface macros to provide C and C++ development environment support (which we will discuss in a moment), so we need to be aware of these macros so that we can derive our interface using them too.

The next code listing is for ID3DXAllocateHierarchy as declared in d3dx9Anim.h.

```
DECLARE_INTERFACE( ID3DXAllocateHierarchy )
{
    // ID3DXAllocateHierarchy
    STDMETHOD(CreateFrame)        ( THIS_ LPCSTR Name,  LPD3DXFRAME *ppNewFrame) PURE;

    STDMETHOD(CreateMeshContainer)( THIS_ LPCSTR Name,
                                    LPD3DXMESHDATA pMeshData,
                                    LPD3DXMATERIAL pMaterials,
                                    LPD3DXEFFECTINSTANCE pEffectInstances,
                                    DWORD NumMaterials,
                                    DWORD *pAdjacency,
                                    LPD3DXSKININFO pSkinInfo,
                                    LPD3DXMESHCONTAINER *ppNewMeshContainer) PURE;

    STDMETHOD(DestroyFrame)       ( THIS_ LPD3DXFRAME pFrameToFree ) PURE;

    STDMETHOD(DestroyMeshContainer)( THIS_ LPD3DXMESHCONTAINER pMeshContToFree ) PURE;
};
```

> **Note:** If you ever have occasion to browse through any of the DirectX header files, you will see that virtually all interfaces are declared just like C++ classes. The difference is that they never have member variables, only member functions. This is exactly what an interface is (for those of you who don't know). The interface exposes only methods to the user of the API and keeps the data and private functionality nicely tucked away. Refer back to our COM discussion in Chapter Two for a refresher if needed.

The declaration may look a little strange, but do not be distracted by the macros, which will be built out during our compiler's pre-process. We will discuss those macros in the next section and it will all become clear. First we will look at how to derive our own class from this interface, which is something we *must* do. It is this derived object that we will pass to the D3DX loading function. D3DX will use the methods of our class to allocate the frames and mesh containers it created in application memory. This allows the application to chose how and where these resources should be allocated.

As the following code shows, deriving a class from an interface is quite similar to normal inheritance models.

```
class CAllocateHierarchy: public ID3DXAllocateHierarchy
{
public:

    // Constructor
    CAllocateHierarchy( int var ) : m_TestVar(var) {}

    STDMETHOD(CreateFrame)           ( THIS_ LPCTSTR Name,   LPD3DXFRAME *ppNewFrame);

    STDMETHOD(CreateMeshContainer)   ( THIS_ LPCTSTR Name,  LPD3DXMESHDATA pMeshData,
                                       LPD3DXMATERIAL pMaterials,
                                        LPD3DXEFFECTINSTANCE pEffectInstances,
                                       DWORD NumMaterials, DWORD *pAdjacency,
                                       LPD3DXSKININFO pSkinInfo,
                                        LPD3DXMESHCONTAINER *ppNewMeshContainer );

    STDMETHOD(DestroyFrame)          ( THIS_ LPD3DXFRAME pFrameToFree );

    STDMETHOD(DestroyMeshContainer)  ( THIS_ LPD3DXMESHCONTAINER pMeshContainerBase );

    // Public Variables for This Class
    int   m_TestVar
};
```

Our derived class adds a constructor and a public member variable (m_TestVar). The four functions of the ID3DXAllocateHierarchy base class have the ‘PURE’ macro at the end of each function prototype. This macro expands to ‘=0’, which means the function is pure virtual and must be implemented in the derived class. That is all there is to deriving from an interface. In truth, this is not something that you will usually want or need to do, but it is required in this particular case. Of course, we do have to implement these four methods in our derived class to provide memory allocation and deallocation functions that D3DX can use during hierarchy construction. We will discuss the implementation of these four methods shortly. If you would like to better understand the macros seen above, read the next section. It is not vital that you understand these macros in order to use them, but we will discuss them anyway for those students who are interested. For those who are not, feel free to skip this next section on move straight on to discuss implementing the four methods of our ID3DXAllocateHierarchy derived class.

9.4.1 The COM Macros

When deriving a class from the ID3DXAllocateHierarchy interface, our derived class methods seem to be using quite a few strange looking macros: STDMETHOD and THIS_ for example. Technically speaking, we do not have to use them in order to derive a class from an interface, but it does make life easier when matching prototypes for base class functions. We want to ensure that they overload correctly in our derived class.

Let us start by looking at what these macros resolve to in the pre-processor at compile time. They can be found in the header file ‘objbase.h’ and are part of the standard COM development semantics. As you might suspect, DirectX follows the COM development guidelines and uses these macros throughout its interface declarations.

When we call the STDMETHOD macro, we pass in the name of the class method like so:

```
STDMETHOD( method )
```

When the pre-processor encounters this macro it resolves to:

```
virtual HRESULT STDMETHODCALLTYPE method
```

This expanded macro in turn contains the 'STDMETHODCALLTYPE' macro, which resolves to:

```
__stdcall
```

This declares the standard COM function calling convention. That is, the STDMETHOD fully resolves to the following, where 'method' is the name of the function passed into the macro.

```
virtual HRESULT _stdcall method
```

Thus, the CreateFrame function for our CAllocateHierarchy test class would be defined like so when processed by the compilers pre-processor:

```
virtual HRESULT _stdcall CreateFrame
```

So the macro is used to quickly define overridable interface functions that return an HRESULT. This is the standard COM style as we learned way back in Chapter Two.

The other macro we see used in the ID3DXAllocateHierarchy interface declaration (and every COM interface declaration) is the 'THIS_' macro. It is inserted before the first parameter of each method. This macro makes it possible to use the interface in a C environment that would otherwise have no concept of objects or methods. There are two versions of the 'THIS_' macro, wrapped by a compile time directive. If we remove some of the bits that are not relevant to this discussion, we end up with the following:

```
#if defined(__cplusplus) && !defined(CINTERFACE)
      #define     THIS_
#else
      #define     THIS_       INTERFACE FAR *This,
#endif
```

This is actually quite clever and is not something you will likely see very often, but it is worth discussing.

All of the macros (such as DECLARE_INTERFACE, STDMETHOD, etc.) allow for C style interface declaration and expansion, even if the application is being compiled in a C++ environment. Although all of these macros have been around forever, and are actually defined by the underlying COM object mechanism (objbase.h), DirectX follows them strictly, to allow a C environment to use the COM objects exposed.

As we can see, the expansion of the above 'THIS_' macro, in a C++ compiler (ignoring the && case for a moment), would result in absolutely nothing being injected into the portions of code by the pre-

processor. In other words, nothing would be inserted before the first parameter of each method. We know that with C++ classes, the 'this' pointer is implicitly inserted as the first parameter of all class member functions by the compiler. In a C environment however, this is not the case. This leads to the second case above, where a 'This' pointer is injected by the macro pre-processor during a compile as the first parameter to each method. But how does it know what type the 'This' parameter inserted into the method parameter list should be? We can see right at the start of the inserted code that there is another macro being used: `INTERFACE`. What we also see, is that at the start of every interface declaration in DirectX (or more specifically in this example, just prior to the declaration of the `ID3DXAllocateHierarchy` interface) the following lines of code which assigns a value to the INTERFACE definition:

```
#undef      INTERFACE
#define     INTERFACE   ID3DXAllocateHierarchy
```

If we look at the expansion of the `CreateFrame` member of the ID3DXAllocateHiearchy interface:

```
STDMETHOD (CreateFrame)( THIS_ LPCSTR Name,  LPD3DXFRAME *ppNewFrame)   PURE;
```

in a C environment, we get the following:

```
virtual HRESULT __stdcall CreateFrame(ID3DXAllocateHierarchy FAR *This,
                                      LPD3DXFRAME * ppNewFrame) = 0;
```

As you can see, as the first parameter to each method, a pointer to the actual structure that contains the member variables is inserted so the function can access them in C. This is exactly what the COM object exports, allowing us to use DirectX in a C environment. There is of course a little extra work to do, because we must remember to always explicitly pass the 'this' pointer when calling an interface method and we have to address some of the C++ specific concepts like 'virtual' and '=0'.

If you look at any of the DirectX header files that declare interfaces you will see, before the DECLARE_INTERFACE macro, the lines

```
#undef      INTERFACE
#define     INTERFACE   NameOfInterface
```

This means that when we use the 'THIS_' macro in a 'C' environment, a far pointer of the correct interface type is inserted at the head of each method's parameter list. You will see these lines of code before each interface declaration where 'NameOfInterface' would of course be replaced by the interface about to be declared. This allows for the behavior of the _THIS macro to be altered on a per interface basis.

Looking again at the compile time directives for the '`THIS_`' macro, we see the "`&& !defined(CINTERFACE)`" portion of the first if statement. This define is available for us to use from inside our C++ environment. We can '`#define CINTERFACE`' right at the start of the logical compile path, and develop using the C language thereafter even when using a C++ compiler. This is a nice additional level of portability.

Before moving on, there is one last issue to cover: how to access interfaces or classes in a C program when the language does not support these concepts. The key is the '`DECLARE_INTERFACE`' macro.

As you know, C++ defines structs and classes similarly. One might consider a struct in C++ as the equivalent of a public class (classes are private by default). Since structs in C are data containers that cannot include member functions, the DECLARE_INTERFACE macro helps bridge the gap.

DirectX uses the '`DECLARE_INTERFACE`' macro wherever it needs to declare an interface. If we imagine we declare a simple interface called ID3DXAllocateHierarchy with a single member function, it would be declared in the DirectX header files something like this:

```
DECLARE_INTERFACE( ID3DXAllocateHierarchy )
{
   STDMETHOD(CreateFrame) (THIS_ LPCSTR Name, LPD3DXFRAME * ppNewFrame) PURE;
};
```

If we look in objbase.h (in your C++ include directory) you will see that the interface declaration shown above is expanded differently depending on whether we are using a C or C++ environment.

```
#if defined(__cplusplus) && !defined(CINTERFACE)

      #define interface                                  struct

      #define DECLARE_INTERFACE(iface)                   \
        interface iface

      #define DECLARE_INTERFACE_(iface, baseiface)       \
              interface iface : public baseiface

            #define STDMETHOD(method)                          \
                    virtual HRESULT STDMETHODCALLTYPE method

            #define PURE                                       = 0

#else
      #define interface                                  struct

      #define DECLARE_INTERFACE(iface)                   \
        typedef interface iface {                  \
          struct iface##Vtbl FAR*lpVtbl;           \
        } iface;                                   \
        typedef struct iface##Vtbl iface##Vtbl; \
        struct iface##Vtbl

      #define DECLARE_INTERFACE_(iface, baseiface)       \
              DECLARE_INTERFACE(iface)

      #define STDMETHOD(method)                          \
              HRESULT (STDMETHODCALLTYPE * method)

      #define PURE

#endif
```

Notice that there are two separate DECLARE_INTERFACE macros, one with an underscore at the end, and one without. The one with the underscore is used wherever a base interface is required.

The above conditional code shows that in the C++ environment, the DECLARE_INTERFACE macro is simply replaced with the keyword 'struct' and, together with the macros used to declare the method (_THIS and STDMETHOD), would expand to a structure called ID3DXAllocateHierarchy which has a single virtual function as its only method. It is a regular structure that has a virtual function which uses the standard COM calling convention and returns an HRESULT as shown below:

```
struct ID3DXAllocateHierarchy
{
   virtual HRESULT __stdcall CreateFrame(LPCSTR Name,LPD3DXFRAME *ppNewFrame) = 0;
};
```

This is perfectly legal C++ and by studying the expansion code above you can see how this structure definition was arrived at.

If a C environment is being used, the DECLARE_INTERFACE macro is expanded to something a bit more cryptic: This is because structures can not have methods in C, so a workaround is put in place. The C section of the expansion would create the following:

```
typedef struct ID3DXAllocateHierarchy
{
  struct ID3DXAllocateHierarchyVtbl FAR * lpVtbl;
} ID3DXAllocateHierarchy;

struct ID3DXAllocateHierarchyVtbl
{
  HRESULT (__stdcall * CreateFrame ) (ID3DXAllocateHierarchy FAR * This,
                                      LPCSTR Name, LPD3DXFRAME * ppNewFrame);
};
```

The C case creates two structures instead of one. The first is the actual ID3DXAllocateHierarchy structure and it contains a single member variable. This is a pointer to the second structure type created by the macro (called ID3DXAllocateHierarchyVtbl). This structure is referred to as the VTABLE and as you can see, it contains one or more pointers to functions as its members. The names of the structures are the same with the exception of the Vtbl appended to the end of the second structure. The Vtable is essentially where functions that were part of the class in the C++ case will be stored. If we were to carry on declaring methods in our interface, this second structure would end up containing callbacks for all of the interface methods. Therefore, we can think of the Vtable structure as being linked to the first structure as a container of function pointers.

In C++ the methods of this interface can be called like so:

```
pAllocHierarchy->CreateFrame (…) ;        // Calling interface methods in C++
```

In C, the function pointers are stored in the Vtable structure which is accessible via the first structure's lpVtbl member. Therefore, we must take this into account when calling member functions of the interface in the C development environment.

```
pAllocHierarchy->lpVtbl->CreateFrame (…); // Calling interface methods in C
```

Inheritance is not implicitly or automatically supported in a C environment. We see this in the **DECLARE_INTERFACE** macro. The two sections (C/C++) of the DECLARE_INTERFACE_ macro (note the underscore) are used to declare an interface that has a base class. The two macros are shown again next:

The C++ case:

```
#define DECLARE_INTERFACE_(iface, baseiface )interface iface : public baseiface
```

This resolves the standard class/struct inheritance code for a declaration:

```
struct iface : public baseiface
```

The C case ignores the base interface:

```
#define DECLARE_INTERFACE_(iface, baseiface)  DECLARE_INTERFACE(iface)
```

This resolves to:

```
struct iface
```

This is unfortunate, but is something that we certainly can live without. All DirectX interfaces re-declare their base class functions anyway, so they are explicitly supported in derived interfaces. We still get access to all of the base class functionality via the VTable callbacks, even if we cannot cast between types in ways that we might take for granted in C++. If you have ever wondered why all of the interfaces in the DirectX header files re-declare all of their base class methods (which should automatically be inherited from the base interface), now you know -- it is for C environment access compatibility where method inheritance is not supported.

Bear in mind that the compiler will automatically assume that it is to compile code for a C environment when the '.c' file extension is used. So be aware of this if you use DirectX in a project which mixes '.c' files, with '.cpp' files.

9.4.2 The ID3DXAllocateHierarchy Interface

The fourth parameter to the D3DXLoadMeshHierachyFromX function is a pointer to an application defined class derived from the ID3DXAllocateHierarchy interface. The derived class must implement the four functions of the base interface so that D3DXLoadMeshHierarchyFromX can call them when mesh containers and frames need to be allocated and de-allocated. This process of calling back out to the application allows for application resource ownership during hierarchy construction. This is important for applications that want to add additional functionality to the interfaces that are provided.

Let us look at an example of an interface derived class:

```
class CAllocateHierarchy: public ID3DXAllocateHierarchy
{
public:

    STDMETHOD(CreateFrame)          ( THIS_ LPCTSTR Name,  LPD3DXFRAME *ppNewFrame);

    STDMETHOD(CreateMeshContainer)  ( THIS_ LPCTSTR Name,  LPD3DXMESHDATA pMeshData,
                                      LPD3DXMATERIAL pMaterials,
                                      LPD3DXEFFECTINSTANCE pEffectInstances,
                                      DWORD NumMaterials, DWORD *pAdjacency,
                                      LPD3DXSKININFO pSkinInfo,
                                      LPD3DXMESHCONTAINER *ppNewMeshContainer );

   STDMETHOD(DestroyFrame)          ( THIS_ LPD3DXFRAME pFrameToFree);

   STDMETHOD(DestroyMeshContainer)  (THIS_ LPD3DXMESHCONTAINER pMeshContainerBase);
};
```

Because this is a class and not merely an interface (a pure abstract base class) we can now add any number of additional methods and member variables to enhance the functionality of the original ID3DXAllocateHierarchy. The above example does not add any new member variables or methods, but it must implement the four base class methods. This allows D3DXLoadMeshHierarchyFromX to use them like callbacks.

9.4.2.1 Allocating Frames

HRESULT CreateFrame(LPCSTR *Name*, LPD3DXFRAME **ppNewFrame*)

This function is called when a new frame is encountered in the X file. The application is responsible for allocating a new D3DXFRAME and then returning it to D3DXLoadMeshHierrachyFromX by means of the second parameter. D3DX will populate the frame we return with data from the X file and attach it to the hierarchy in the correct position. At its simplest, this function just has to allocate a new D3DXFRAME structure and point the second parameter passed in to point at it.

LPCSTR Name
This first parameter is the name of the frame being created and will be passed into the function by D3DXLoadMesHierarchyFromX when it encounters a frame in the X file that is about to be loaded. Recall that there is storage for a string pointer in the base frame structure. In the case of a character model for example, the artist or 3D modeller might name the frames with descriptive labels such as “Left Arm”, “Right Leg” or “Torso” within the X file.

When we allocate the D3DXFRAME structure inside this function, we will want to store this name in the D3DXFRAME::Name member variable so that we can access the frame by name later. However it is important to realize that the memory of the string we are passed is not owned by the application, it is owned by D3DX. As such, we cannot simply store the string pointer we are passed in our D3DXFRAME structure because we cannot make assumptions about what D3DX will do with the memory once we return from the function. For example, if D3DX were to free it, our frame structure

would contain a dangling pointer to a string that no longer exists. Therefore this string must be duplicated in any way you see fit (i.e. strdup or _tcsdup if you want to be Unicode compliant) and stored in the D3DXFRAME structure you create. Remember that if you duplicate the name, you must make sure that when the frame is destroyed you also free the string memory that you created during the frame creation process. We will examine how frames are destroyed in a moment.

LPD3DXFRAME *ppNewFrame

The second parameter to this function is the address of a D3DXFRAME pointer (i.e. a pointer to a pointer). This time the application is being passed a pointer by D3DX, not the other way round as is usually the case with D3DX function calls. The de-referenced pointer should contain `NULL` at this point, although it may not. We use this pointer to return the pointer to the frame that we create, back to the D3DXLoadMeshHierarchyFromX function.

The following code snippet is a simple version of a CreateFrame function implementation. It does everything D3DX requires and a bit more. It allocates a new D3DXFRAME, copies the name passed into the function, sets the frame transformation matrix to an identity matrix, and assigns the passed pointer to the newly created frame. Setting the transformation matrix to identity is purely a safety precaution as it will most likely be overwritten with the matrix data extracted from the X file when the frame data is loaded. However, this is a nice default state.

```
HRESULT CAllocateHierarchy::CreateFrame( LPCTSTR Name, LPD3DXFRAME *ppNewFrame )
{
    D3DXFRAME  *pNewFrame = NULL;

    // Clear out the passed frame pointer (it may not be NULL)
    *ppNewFrame = NULL;

    // Allocate a new frame
    pNewFrame = new D3DXFRAME;
    if ( !pNewFrame ) return E_OUTOFMEMORY;

    // Clear out the frame
    ZeroMemory( pNewFrame, sizeof(D3DXFRAME_MATRIX) );

    // Store the passed name in the frame structure
    // and set the new frame transformation matrix
    // to an identity matrix by default
    if ( Name ) pNewFrame->Name = _tcsdup( Name );
    D3DXMatrixIdentity( &pNewFrame->TransformationMatrix );

    // Assign the D3DX passed pointer to our new application allocated frame
    *ppNewFrame = pNewFrame;

    // Success!!
    return D3D_OK;
}
```

When we examine how to derive from these structures a little later on, you will see that these Create functions can and do serve a greater purpose, allowing us to store other data in our frame objects. When the above function returns, the passed pointer will point to our application allocated frame structure and it will contain the frame's name and an identity matrix. D3DX will then load the frame data from the X

file for this frame and store it in the structure before inserting the frame into its correct place in the hierarchy. This function will be called once for every frame in the hierarchy.

9.4.2.2 Deallocating Frames

If D3DX does not own the frame hierarchy memory, then how can it deallocate that memory when the hierarchy needs to be destroyed? In fact, since our application will only ultimately get back a root frame pointer from the D3DXLoadMeshHierarchyFromX function, how can we ourselves deallocate the hierarchy?

It just so happens that the D3DX library exposes a global function called D3DXFrameDestroy that will help with this process. When we have finished with our frame hierarchy we will call this function to cleanup the memory. We will discuss this function in detail at the end of this section. For now you should know that the D3DXFrameDestroy call takes two parameters. This first parameter is a pointer to a frame. This input frame and all of its descendants in the hierarchy will be destroyed. This would include the deallocation of any meshes as well. To release the whole hierarchy, we would pass in the root frame; to release only a sub-tree in the hierarchy, we would pass the appropriate node.

Because the memory is application owned, the application is responsible for actually physically releasing that memory. The second parameter to the D3DXFrameDestroy function is a pointer to an ID3DXAllocateHierarchy derived object (just as we saw for allocation). The function will step through each frame in the hierarchy and call ID3DXAllocateHierarchy::DestroyFrame. This gives control back to the application defined class for memory release.

HRESULT DestroyFrame(LPD3DXFRAME *pFrameToFree*);

The single function parameter is a pointer to the frame that D3DX would like released. This function will be called once for every frame in the hierarchy when the hierarchy is destroyed. The destruction of the full hierarchy is set in motion by the application calling the global D3DXFrameDestroy function with a pointer to the root frame.

Next we see a quick example implementation that matches up with the CreateFrame example we wrote earlier. Notice how the function frees the passed frame and the frame name as well. The name was allocated in the CreateFrame function when _tcsdup was called to make a copy of the passed name. Just as the CAllocateHierarchy::CreateFrame function is called by D3DX for each frame that needs to be created during X file loading, the CAllocateHierarchy::DestroyFrame function is called when the memory for each frame in the hierarchy needs to be released.

```
HRESULT CAllocateHierarchy::DestroyFrame( LPD3DXFRAME pFrameToFree )
{
     // Validate Parameters
    if ( !pFrameToFree ) return D3D_OK;

    // Release memory for name and frame
    if ( pFrameToFree->Name ) free( pFrameToFree->Name ); // '_tcsdup' allocated.
    delete pFrameToFree;                                  // 'new' allocated.
```

```
    // Success!!
    return D3D_OK;
}
```

Note that we used ‘free’ and not ‘delete’ to free the string memory because the _tcsdup function used in the CreateFrame method uses a C memory allocation function (alloc or malloc, etc.). The frame itself was allocated using ‘new’, so we free it with ‘delete’.

9.4.2.3 Allocating Mesh Containers

To handle allocation of meshes and related resources stored in the X file, our application-derived class must override the ID3DXAllocateHierarchy::CreateMeshContainer function. This function is similar to CreateFrame. It is called by D3DX so that the application can allocate and initialize a D3DXMESHCONTAINER structure and return it to the file loading function. Recall that D3DX can automate the loading of the mesh geometry, but it does not manage resources such as materials, textures, or effects (the application is responsible for that task). We saw this in the last chapter when we loaded the textures for the passed texture filenames and stored them in our CScene class.

The initialization and validation code is a bit laborious -- we have to copy, validate, and potentially manipulate all of the mesh related data passed in before we return the mesh container to D3DX. The concept is very similar to what we did in the last chapter when loading meshes using D3DXLoadMeshFromX. Recall that we are passed a loaded mesh and buffers filled with mesh related assets such as materials and texture filenames and we must process those resources ourselves. This function will be called once for each mesh in the X File hierarchy.

```
HRESULT CreateMeshContainer
(
    LPCSTR           Name,
    LPD3DXMESHDATA   pMeshData,
    LPD3DXMATERIAL   pMaterials,
    LPD3DXEFFECTINSTANCE pEffectInstances,
    DWORD            NumMaterials,
    DWORD *          pAdjacency,
    LPD3DXSKININFO   pSkinInfo,
    LPD3DXMESHCONTAINER *ppNewMeshContainer
);
```

LPCTSTR Name
This string stores the name of the mesh as defined inside the X file. We will often want to store this name along with our mesh so we can identify it later. Like the name in the frame structure, we must free the name when the mesh container is destroyed.

LPD3DXMESHDATA pMeshData
D3DX passes us the relevant mesh object using this structure pointer. It will either contain an ID3DXMesh, an ID3DXPMesh, or an ID3DXPatchMesh. We can do whatever we like with the mesh - clone it, manipulate it and then store it in the mesh container that we allocate, or we can simply store the passed mesh in the D3DXMESHCONTAINER structure immediately. It is important to understand that

eventually D3DX will Release() the interface to the mesh passed into this function. Therefore, if you are going to store away the mesh interface and you would like it to be persistent, you must call AddRef().

LPD3DXMATERIAL **pMaterials**
LPD3DXEFFECTINSTANCE **pEffectInstances**
DWORD **NumMaterials**
DWORD * **pAdjacency**
LPD3DXSKININFO **pSkinInfo**

These parameters are essentially carbon copies of the information you are required to store in the D3DXMESHCONTAINER structure. They are very similar to the information we get back when we call D3DXLoadMeshFromX. Again, we do not own the memory for these input parameters, and they will be destroyed by the caller (D3DX), so we must duplicate them if we wish to store the resources in the mesh container.

LPD3DXMESHCONTAINER * ppNewMeshContainer

As with the CreateFrame function, this pointer allows us to return our newly allocated mesh container back to D3DX after it has been populated with data passed into the function that we want to retain. For now, let us just look at a quick example that deals with the basics of mesh allocation. We will look at a more complete implementation that handles materials and textures in the workbook accompanying this chapter.

```
HRESULT CAllocateHierarchy::CreateMeshContainer( LPCTSTR Name, LPD3DXMESHDATA pMeshData,
                                                 LPD3DXMATERIAL pMaterials,
                                                 LPD3DXEFFECTINSTANCE pEffectInstances,
                                                 DWORD NumMaterials, DWORD *pAdjacency,
                                                LPD3DXSKININFO pSkinInfo,
                                                 LPD3DXMESHCONTAINER *ppNewMeshContainer )
{
    HRESULT            hRet;
    LPD3DXMESH         pMesh = NULL;
    D3DXMESHCONTAINER *pMeshContainer = NULL;

    // We only support standard meshes in this example (no patch or progressive meshes)
    if ( pMeshData->Type != D3DXMESHTYPE_MESH ) return E_FAIL;

    // We require FVF compatible meshes only
    if ( pMesh->GetFVF() == 0 ) return E_FAIL;

    // Allocate a new mesh container structure
    pMeshContainer = new D3DXMESHCONTAINER_DERIVED;
    if ( !pMeshContainer ) return E_OUTOFMEMORY;

    // Clear out the structure to begin with
    ZeroMemory( pMeshContainer, sizeof(D3DXMESHCONTAINER_DERIVED) );

    // Copy over the name. We can't simply copy the pointer here because the memory
    // for the string belongs to the caller (D3DX)
    if ( Name ) pMeshContainer->Name = _tcsdup( Name );

    // Copy over passed mesh data structure into our new mesh contain 'MeshData' member
    // Remember, we are also copying the mesh interface stored in the 'MeshData' structure
    // so we should increase the ref count
    pMeshContainer->MeshData = *pMeshData;
    pMeshContainer->MeshData.pMesh->AddRef();
```

```
    // Attempt to optimize the new mesh
    pMeshContainer->MeshData.pMesh->OptimizeInPlace(D3DXMESHOPT_VERTEXCACHE ,
                                                    pAdjacency , NULL , NULL , NULL);

    // Store this new mesh container pointer
    *ppNewMeshContainer = (D3DXMESHCONTAINER*)pMeshContainer;

    // Success!!
    return D3D_OK;
}
```

Studying the previous example function shows us that the process is not very complicated. It basically just creates a new mesh container and copies the passed mesh into it. The input pointer is assigned so that D3DX can insert the mesh into the hierarchy. Processing materials and textures will make the function longer, but no more difficult. The same is true if we need to change the mesh format to accommodate data required by the application, such as vertex normals if they are not included in the X file. In this case we would simply clone the passed mesh and store the newly cloned mesh in the mesh container. When this function returns the newly created mesh container to D3DX, it will be attached to the correct parent frame in the hierarchy. This function will be called for every mesh contained in the X file.

9.4.2.4 Deallocating Mesh Containers

During hierarchy destruction, ID3DXAllocateHierarchy::DestroyMeshContainer is called by D3DXFrameDestroy for each mesh container in the hierarchy. This allows the application to free the mesh containers and any application-defined content in an appropriate way. This parallels the requirement to implement a DestroyFrame function for frame cleanup.

`HRESULT DestroyMeshContainer( LPD3DXMESHCONTAINER pMeshContToFree );`

DestroyMeshContainer takes a single parameter: a pointer to the mesh container that D3DX would like us to free from memory. Here is an example that works in conjunction with the CreateMeshContainer function just discussed:

```
HRESULT CAllocateHierarchy::DestroyMeshContainer(LPD3DXMESHCONTAINER pContToFree)
{
    // Validate Parameters
    if ( !pContToFree ) return D3D_OK;

    // Release data
    if ( pContToFree->Name ) free( pContToFree->Name );
    if ( pContToFree->MeshData.pMesh)
         pContToFree->MeshData.pMesh->Release();

    delete pContToFree;

    // Success!!
    return D3D_OK;
}
```

9.4.2.5 The D3DXFrameDestroy Function

Most of the resource deallocation was handled in the DestroyFrame and DestroyMeshContainer functions we just examined. Initiating the destruction of the hierarchy, or a subtree thereof, is done with the following global D3DX function:

```
HRESULT D3DXFrameDestroy
(
    LPD3DXFRAME pFrameRoot,
    LPD3DXALLOCATEHIERARCHY pAlloc
);
```

LPD3DXFRAME pFrameRoot
This frame is the starting point for the destruction process. D3DX will start deallocation here at this node and then work its way through all subsequent child frames and mesh containers, cleaning up the resources as it goes. To destroy the entire hierarchy, just pass the root frame that was returned via the call to `D3DXLoadMeshHierarchyFromX`. Be sure to set your root frame variable to `NULL` after calling this function, to ensure that you do not try to access invalid memory later on.

LPD3DXALLOCATEHIERARCHY *pAlloc*
The second parameter is an instance of our derived `ID3DXAllocateHierarchy` class. It does not have to be the same physical object that was passed into the D3DXLoadMeshHierarchyFromX function, and it does not have to be allocated in any special way. For example, this object could be temporarily instantiated on the stack and it will work just as well. This will be the object D3DX calls to access the two Destroy functions (DestroyFrame and DestroyMeshContainer). Be sure to pass in the same type of derived class that was used to create the hierarchy in the first place.

Here is an example of a call that creates a local allocator and then cleans up the entire hierarchy:

```
CAllocateHierarchy Allocator;

if (m_pFrameRoot) D3DXFrameDestroy(m_pFrameRoot, &Allocator);
m_pFrameRoot = NULL;
```

9.4.3 Extending the D3DX Hierarchy

At first it may seem that hierarchy loading is more complicated than is called for. After all, D3DX should certainly be perfectly capable of doing everything itself. But by managing much of the process at the application level, we gain a significant advantage: the ability to extend the data structures stored in the hierarchy.

D3DX will concern itself only with the core navigation components of the hierarchy (`D3DXFRAME::pFrameSibling`, `D3DXFRAME::pFrameFirstChild`, `D3DXFRAME::pMeshContainer` and `D3DXMESHCONTAINER::pNextMeshContainer`), the frame transformation matrix (when working with

animation) and the frame name (when working with skinned meshes). The rest of the data structure can be used at our discretion.

As a result, we can derive our own structures from both `D3DXFRAME` and `D3DXMESHCONTAINER` and store whatever additional information we require. Since we are responsible for both the creation and destruction of these structures, as long as we cast it correctly in all of the right places, D3DX will work correctly without requiring knowledge of our additional structure members.

There are a number of cases where this proves to be useful. Some examples would include:

1) Caching the world matrix. Because the frame matrices are relative to their respective parent's space, they can be quite difficult (and potentially expensive) to work with when it comes to rendering the hierarchy. This is especially true in cases where multiple frames exist at the same level (as a sibling linked list).

   To expedite the process, each derived frame can store a new matrix that is a combination of all parent frame transformation matrices and its own relative matrix. We can build these matrices in a single hierarchy update pass. Then, when we render each mesh in the hierarchy, we can just use this matrix (assigned to its owner frame) as the mesh world matrix and avoid having to combine relative matrices for each mesh that we render.

   So we might derive a structure from D3DXFRAME similar to the following:

```
struct D3DXFRAME_DERIVED : public D3DXFRAME
{
      D3DXMATRIX  mtxCombined;
};
```

2) Hierarchical intersection testing. We can add bounding volume information to our frame structures so that hierarchical tests can be performed. For example, we might choose to store an axis aligned bounding box for collision testing, frustum culling, etc.:

```
struct D3DXFRAME_DERIVED : public D3DXFRAME
{
      D3DXMATRIX  mtxCombined;
      D3DXVECTOR3 aabb_min, aabb_max;
};
```

   The bounding volume at each frame in the hierarchy would be a combination of the frame's immediate bounding volume (built using all of the bounding volumes for meshes stored in the frame) plus the bounding volumes for its list of direct children (which themselves are calculated in the same way). These bounding volumes would all be world space volumes, and the bounding volume hierarchy would be built from the bottom up. If a node animates or changes in some way that would affect position, orientation, or scale, we can just recalculate its bounding volume and propagate that change up the tree to the root, starting from the node that changed. The significant advantage here is that the parent bounding volume totally encloses its children. Thus, if the parent is not visible (via an AABB/Frustum test for example) then none of its children could possibly be visible either. This minimizes rendering calls and is a common optimization in 3D

engines. The same holds true for other collision tests as well. If something cannot collide with the parent volume, it cannot possibly collide with any of its children either. This is one of the most beneficial aspects of a spatial hierarchy and we will encounter it again and again in our 3D graphics programming studies (including later in this very course).

3) Application specific mesh management. Rather than use the provided D3DXMESHDATA structure, we can derive our own mesh container to suit our needs:

```
struct D3DXMESHCONTAINER_DERIVED : public D3DXMESHCONTAINER
{
      CTriMesh    *pMesh;
};
```

Here we can store the mesh as we require and work directly within our engine's pre-established framework. This allows us to plug our mesh class directly into the D3DX hierarchy and store it in a mesh container. The CTriMesh listed above would be an example of a proprietary application mesh class that you might write to manage your triangle based mesh data.

One thing to make sure that you do straight away is update the destroy functions in your hierarchy allocation class. If you are working with a derived structure, you must make sure that you cast the pointer that was passed to the destroy functions to the derived structure type before you `delete` it. If you do not, you risk important destructors not being called, memory not being freed, or any manner of potential general protection faults.

There are some important exceptions to the above cases, where overriding the default behaviour may cause problems. For example, not storing the mesh in the provided D3DXMESHDATA structure can lead to one issue. Consider the function `D3DXFrameCalculateBoundingSphere`. This is a very nice function that iterates through the frame hierarchy starting from a passed frame and tests the mesh data stored there to return a bounding sphere that encapsulates all of the meshes that are children (both immediate and non-immediate). This function will expect the mesh data to be stored in the standard place (inside the D3DXMESHDATA structure of the mesh container) and will not be able to calculate a bounding sphere unless this is the case. Another example is `D3DXSaveMeshHierarchyToFile`. This function iterates through the hierarchy and writes mesh, material, effect, and texture data out to an X file. Note that in both of these cases there is a simple solution -- we can store a reference to the mesh in both the provided data structure (D3DXMESHDATA) and our custom mesh class (CTriMesh in our example above).

9.4.4 D3DXLoadMeshHierarchyFromX Continued

There are still two parameters left to discuss before we use D3DXLoadMeshHierarchyFromX.

```
HRESULT D3DXLoadMeshHierarchyFromX
(
    LPCTSTR                 Filename,
    DWORD                   MeshOptions,
    LPDIRECT3DDEVICE9       pDevice,
```

```
    LPD3DXALLOCATEHIERARCHY       pAlloc,
    LPD3DXLOADUSERDATA            pUserDataLoader,
    LPD3DXFRAME*                  ppFrameHeirarchy,
    LPD3DXANIMATIONCONTROLLER*    ppAnimController
);
```

We will address the animation controller type in the next chapter. For now, we will just assume that the X file contains no animation and we will set this parameter to NULL. This tells the function that we are not interested in getting back animation data, even if it exists in the file.

LPD3DXLOADUSERDATA pUserDataLoader

The fifth parameter is a pointer to an ID3DXLoadUserData interface. Much like ID3DXAllocateHierarchy, this parameter is also going to be passed as an application-defined class that is derived from the ID3DXLoadUserData interface. The methods of this object will be called (by the D3DX loading function) whenever data objects built from custom templates are encountered in the X file.

If you are not using custom data objects, then D3DX will not require this pointer and you can simply pass NULL. In fact, even if the X file does contain custom data objects you can still pass in NULL. In this case, D3DX will simply ignore custom information and just load the standard data objects.

Our ID3DXLoadUserData derived object must implement three functions: LoadTopLevelData, LoadFrameChildData, and LoadMeshChildData. In order to properly derive from this class and process custom data objects, we must be familiar with all the X file topics we discussed earlier. Specifically, we need to be comfortable with the IDirectXFileData interface. While we will not need to use this interface, D3DX will pass our callback functions an ID3DXFileData interface for each custom data object it encounters. This interface is almost identical to the IDirectXFileData interface we discussed earlier. It is a newer interface however that has been wrapped up in the D3DX library to make accessing the data a little simpler. If you understand the IDirectXFileData methods, you will understand how to use all of the analogous methods exposed by the ID3DXFileData interface.

9.4.4.1 The ID3DXLoadUserData Interface

The ID3DXLoadUserData interface is declared in d3dx9anim.h. This is another pure abstract non-COM interface just like ID3DXAllocateHierarchy. However, if we look at the following interface declaration we will notice something different about this declaration when compared to all others.

```
DECLARE_INTERFACE(ID3DXLoadUserData)
{
    STDMETHOD(LoadTopLevelData)   ( LPD3DXFILEDATA pXofChildData) PURE;

    STDMETHOD(LoadFrameChildData) ( LPD3DXFRAME pFrame, LPD3DXFILEDATA pXofChildData) PURE;

    STDMETHOD(LoadMeshChildData)  ( LPD3DXMESHCONTAINER pMeshContainer,
                                    LPD3DXFILEDATA pXofChildData) PURE;
};
```

Compare this to the following interface that more closely follows the rules we discussed earlier about declaring interfaces: (It is probably how ID3DXLoadUserData should have been defined.)

```
#undef INTERFACE
#define      INTERFACE ID3DXLoadUserData

DECLARE_INTERFACE(ID3DXLoadUserData)
{
    STDMETHOD(LoadTopLevelData)   ( THIS_ LPD3DXFILEDATA pXofChildData) PURE;

    STDMETHOD(LoadFrameChildData) ( THIS_ LPD3DXFRAME pFrame,
                                    LPD3DXFILEDATA pXofChildData) PURE;

    STDMETHOD(LoadMeshChildData)  ( THIS_ LPD3DXMESHCONTAINER pMeshContainer,
                                    LPD3DXFILEDATA pXofChildData) PURE;
};
```

So there are two key differences (perhaps by design, perhaps an oversight) -- the interface is not defined before the declaration and the methods have not had the 'THIS_' macro inserted at the head of their parameter lists. As a result, this interface cannot be used in a C environment.

In fact, there are two interfaces in this header file (d3dx9anim.h) where the same declaration style is presented. The other is the declaration of the ID3DXSaveUserData interface. If you need to use either interface in a C environment, your only option is to alter the interface declaration in d3dx9anim.h to the format seen above.

Deriving a class from the ID3DXLoadUserData interface in a C++ environment is straightforward:

```
class CLoadUserData : public ID3DXLoadUserData
{
public:
    STDMETHOD(LoadTopLevelData)   (LPD3DXFILEDATA pXofChildData);
    STDMETHOD(LoadMeshChildData)  (LPD3DXMESHCONTAINER pMeshContainer,
                                   LPD3DXFILEDATA pXofChildData);
    STDMETHOD(LoadFrameChildData) (LPD3DXFRAME pFrame, LPD3DXFILEDATA pXofChildData);
};
```

If you are using a C environment and you have edited and corrected the ID3DXLoadUserData declaration in the d3dx9anim.h header file as mentioned above, then your derived class declaration would look something like this:

```
class CLoadUserData : public ID3DXLoadUserData
{
public:
    STDMETHOD(LoadTopLevelData)  (THIS_  LPD3DXFILEDATA pXofChildData);
    STDMETHOD(LoadMeshChildData) (THIS_  LPD3DXMESHCONTAINER pMeshContainer,
                                  THIS_  LPD3DXFILEDATA pXofChildData);
    STDMETHOD(LoadFrameChildData)(THIS_  LPD3DXFRAME pFrame, LPD3DXFILEDATA pXofChildData);
};
```

9.4.4.2 Loading Custom Top Level Data

HRESULT LoadTopLevelData(LPD3DXFILEDATA *pXofChildData*);

Recall that a top level data object is not defined inside any other data object (i.e., it does not have a parent) in the X file. When the D3DXLoadMeshHierarchyFromX function encounters a top level data object that is not one of the data objects based on a standard template, it calls the LoadTopLevelData function of your passed ID3DXLoadUserData derived object so that the application can process the data as it deems appropriate.

The function receives a pointer to an ID3DXFileData interface which provides all of the methods we need to identify the custom object (GetName, GetType, and GetID) and also allows us to retrieve a pointer to the actual data of the object for extraction (via its Lock and Unlock methods). We discussed the IDirectXFileData interface in detail earlier in this chapter and the ID3DXFileData interface is very similar. Its methods are shown below:

```
GetChildren ( SIZE_T *puiChildren );
GetChild    ( SIZE_T uiChild, ID3DXFileData **ppChild );
GetEnum     ( ID3DXFileEnumObject **ppObj );
GetId       ( LPGUID pId );
GetName     ( LPSTR szName, SIZE_T *puiSize );
GetType     ( const GUID *pType );
IsReference ( VOID );
Lock        ( SIZE_T *pSize, const VOID **ppData );
Unlock      ( VOID );
```

To see how we might implement such a function to handle the processing of top level custom data objects, let us go back to our earlier example using a custom template.

```
template TestTemplate
{
  <A96F6E5D-04C4-49C8-8113-B5485E04A866>
  FLOAT TestFloat1;
  FLOAT TestFloat2;
  FLOAT TestFloat3;
}
```

Let us assume that the X file we are loading has at least one of these objects as a top level object. In this example we will include an object called MyFirstValues:

```
TestTemplate MyFirstValues
{
  1.000000;
  2.000000;
  3.000000;
}
```

In order to store and manage this type of data in our application, we would probably define a structure in our code that mirrors the template definition in the X file:

```
struct TestTemplate
{
    float TestFloat1;
    float TestFloat2;
    float TestFloat3;
};
```

Somewhere else in our code, before we load the X file, we would want to define a GUID that mirrors the values stored in the template. This will allow us to compare GUIDs when we need to determine custom object types when the function is called.

```
// The GUID used in the file for the template, so we can identify it
const GUID IID_TestTemplate = {0xA96F6E5D, 0x4C4, 0x49C8, {0x81, 0x13, 0xB5, 0x48,
0x5E, 0x4, 0xA8, 0x66}};
```

You could also create this GUID using the DEFINE_GUID macro:

```
DEFINE_GUID ( IID_TestTemplate , 0xA96F6E5D, 0x4C4, 0x49C8, 0x81, 0x13, 0xB5, 0x48,
0x5E, 0x4, 0xA8, 0x66 };
```

Now let us look at our LoadTopLevelData function:

```
HRESULT CLoadUserData::LoadTopLevelData(LPD3DXFILEDATA pXofChildData)
{
    // First retrieve the name of the data object
    TCHAR DataName[256];
    ULONG StringSize = 256;
    pXofChildData->GetName( DataName, &StringSize );

    // Next retrieve the GUID of the data object if one exists
    GUID DataUID;
    pXofChildData->GetId( &DataUID );

    // Next up, retrieve the GUID of the template which defined it.
    const GUID * pTypeUID;
    pXofChildData->GetType(&pTypeUID );

    // Ensure that we are importing the type that we expect
    if ( memcmp( pTypeUID, &TestTemplateUID, sizeof(GUID) ) != 0 )
        return D3D_OK;

    // OK, we know it's our type, import and process the data
    TestTemplate * pData;
    ULONG DataSize;
    pXofChildData->Lock( &DataSize, ( const void**)&pData );

    // We can now use the data
    DO SOMETHING WITH DATA HERE (ex. COPY AND STORE IT SOMEWHERE);

    PXofChildData->Unlock();

    // Success!!
    return D3D_OK;
}
```

This is the core of our custom data processor. We retrieve the type via a call to `GetType`, the data itself via a call to `GetData`, and then depending on the value of the type `GUID` returned, we cast and process the data pointed to by the returned pointer. If we were using multiple custom templates, a switch statement or some other conditional would probably be in order, but the concept is the same.

Note: Remember that IDirectXFileData::GetID returns the GUID of the object, not the template.

Do not forget that if you are processing data objects that have child data objects (or child data reference objects) then you will want to call IDirectXFileData::GetNextObject to retrieve and process those as well. As the child objects may themselves have child objects, suffice to say, this could be a deeply recursive process which requires some thought on how best to implement this code. Using closed or restricted templates can help minimize code complexity and recursion.

It should be noted that the code example above was written a little bit dangerously in an effort to improve clarity of the process. It assumes that any padding added to the structure TestTemplate by the compiler would also be mirrored in the X file data object, which is not usually the case. Therefore, because file data is not guaranteed to be aligned properly with byte boundaries in your C++ structure, you should access *data* with unaligned pointers. For example, you could step through the data with a BYTE pointer and extract each member by correctly offsetting and casting the de-referenced pointer into another variable type.

Refer to the CD3Dfile.cpp file that ships with the DirectX SDK (Samples\C++\Common\Src) to see how IDirectXFileData can be used to recursively process a hierarchy.

9.4.4.3 Loading Custom Child Data

When a top level object based on one of the open standard templates includes a custom child, D3DX must call out to the application to deal with this data. There are two important functions we must be prepared to implement in our derived ID3DXLoadUserData class. The first function will handle custom children found when processing a frame and the second handles custom children found when processing a mesh.

```
HRESULT LoadFrameChildData(LPD3DXFRAME pFrame,
                           LPD3DXFILEDATA pXofChildData);
```

LoadFrameChildData will be called by D3DXLoadMeshHierarchyFromX whenever an unknown object is encountered by the loading code that is an immediate child of a frame being processed. In addition to the expected ID3DXFileData type that allows us to extract the custom data, the function also takes a pointer to the frame in the hierarchy for which the custom data object is a child. This is useful if our application is using a derived D3DXFRAME structure with additional members that might be needed to store this custom data in some meaningful way. For example, the custom object might contain a string that describes what the frame is used for and whether or not it should be animated. If we were using a derived D3DXFRAME class, we could copy this string into a member variable in the frame structure. Thus each frame would contain a description that could be used for printing out debug information.

The following snippet shows our custom template embedded inside a standard frame object as an immediate child:

```
Xof 0303txt 0032

Frame Root
{
      FrameTransformMatrix
      {
        1.000000, 0.000000, 0.000000,  0.000000,
        0.000000, 1.000000, 0.000000,  0.000000,
        0.000000, 0.000000, 1.000000,  0.000000,
       50.000000, 0.000000, 50.000000, 1.000000;;
      }

      Mesh CarBody { mesh data would go in here }

      TestTemplate MyFirstValues
      {
            1.000000;
            2.000000;
            3.000000;
      }
}
```

When D3DXLoadMeshHierarchyFromX encounters a custom data object as an immediate child of a mesh object, the LoadMeshChildData method is called to facilitate processing. Like its predecessor, the function provides access to the parent mesh in addition to the custom child data.

HRESULT LoadMeshChildData(LPD3DXMESHCONTAINER *pMeshContainer*,
LPD3DXFILEDATA *pXofChildData*);

The following X file snippet demonstrates a custom template embedded in a standard mesh object.

```
Xof 0303txt 0032

Frame Root
{
  FrameTransformMatrix
  {
    1.000000, 0.000000, 0.000000,  0.000000,
    0.000000, 1.000000, 0.000000,  0.000000,
    0.000000, 0.000000, 1.000000,  0.000000,
   50.000000, 0.000000, 50.000000, 1.000000;;
  }

  Mesh MyMesh
  {
    6;                  // Number of vertices
   -5.0;-5.0;0.0;,      // Vertex list ( Vector array )
   -5.0;5.0;0.0;,
    5.0;5.0;0.0;,
    5.0;-5.0;0.0;,
```

```
   -5.0;5.0;-5.0;,
   -5.0;5.0;5.0;;

    2;            // Number of faces
    3;0,1,2;,     // MeshFace Array
    3;3,4,5;;

    TestTemplate MyFirstValues
    {
      1.000000;
      2.000000;
      3.000000;
    }
  } // end mesh
} // end frame ( top level object )
```

9.4.5 Using D3DXLoadMeshHierarchyFromX

Our previous discussions may lead you to believe that D3DXLoadMeshHierarchyFromX is dreadfully complicated. While it is certainly more involved than D3DXLoadMeshFromX, it is still relatively straightforward, especially when you are not processing custom data objects. Once you have derived a suitable ID3DXAllocateHierarchy class to handle frame and mesh allocation, you can load a frame hierarchy stored in a file like so:

```
CAllocateHierarchy Allocator;
D3DXFRAME *pRootFrame = NULL;

D3DXLoadMeshHierarchyFromX("Example.x", D3DXMESH_MANAGED,
                           pDevice, &Allocator,
                           NULL, &pRootFrame, NULL );
```

When this function returns, pRootFrame will point to the root frame of the hierarchy contained in "Example.x". We passed NULL for the pUserDataLoader parameter so that custom data objects will be ignored. We also passed NULL for the ppAnimController parameter since we will not require animation data in this example. We will discuss the ID3DXAnimationController in the next chapter. Note that we have passed D3DXMESH_MANAGED for the mesh creation flag to indicate our desire for managed resources.

> **Note:** The Options passed to D3DXLoadFrameHierarchyFromX are not adhered to as closely as you might like. For example, in one of our testing scenarios, passing D3DXMESH_MANAGED resulted in meshes created with the D3DXMESH_SOFTWARE option (with a dramatic performance hit). To work around this, as you will see in the workbook source code, we forcibly set the Options inside the derived CreateMeshContainer method by cloning the mesh when the options passed by D3DX are not what we asked for.

It is also worth noting that there is a D3DXLoadMeshHierarchyFromXInMemory function available as well. As we have come to expect from the D3DX library, any function that loads data from a file usually has sibling functions that have the same semantics, but different data sources (memory, resources).

9.5 X File Notes

Before wrapping up our X file discussions, there are a few things to keep in mind when working with X files:

- You **must** include the decimal point when working with floating point values in a template. For example, specify '2.' or '2.0' but not '2'. If you do not, the value will be imported incorrectly.

- Be aware of structure padding for byte alignment. This is important to bear in mind when copying data from an IDirectXFileData object into an application defined structure. For example, consider the following structure:

```
struct TestStruct{
    UCHAR foo;
    ULONG bar;
};
```

  You might expect that if we were to do sizeof(TestStruct) that it would return 5, because the structure contains a single ULONG (4 bytes) and a single UCHAR (1 byte). In fact, sizeof would return 8 because by default, structures are byte aligned to make them more efficient. So in memory, our structure looks more like:

```
struct TestStruct{
    UCHAR foo;
    // Three Padding Bytes Here
    ULONG bar;
};
```

  Thus, when laid out in a format similar to the current example, structures are aligned based on the largest member contained therein. Another example:

```
struct TestStruct{
    UCHAR foo;
    // Seven Padding Bytes Here
    __int64 bar;
};
```

  The above structure would return a value of 16 from sizeof, even though we only directly use 9 bytes with our member variables. This is in contrast to our templates -- when D3DX loads a template with members similar to the above structures, it will not pad the data. This means that in some cases we can not simply 'memcpy' the data over, but will need to copy the members individually.

- The SDK documentation states (in places) that restricted template declarators should specify a GUID within square brackets. This is not true -- they should in fact be enclosed in 'less than' and 'greater than' chevron symbols as explained earlier.

9.6 Rendering the Hierarchy

Rendering a D3DX based hierarchy requires stepping through the frames in a particular order. Our function will iterate recursively from frame to frame, combining frame matrices as it goes. We can render meshes stored in the frame in this recursion, or we can alternatively choose to separate the world matrix update pass from the rendering pass. Here is an example of both updating the matrices and rendering in a single pass through the hierarchy:

```
void RenderFrame ( LPD3DXFRAME pFrame, const D3DXMATRIX * pParentMatrix )
{
    // Used to hold world matrix for immediate child meshes of this frame
    D3DXMATRIX WorldMatrix;

    // If there is a parent matrix then combine this frame's relative matrix with it
    // so we get the complete world transform for this frame.
    // Otherwise, this is the root frame so this frame's matrix is
    // the world matrix for the frame (just copy it)
    if ( pParentMatrix != NULL)
      D3DXMatrixMultiply(&WorldMatrix, &pFrame->TransformationMatrix, pParentMatrix);
    else
      WorldMatrix = pFrame->TransformationMatrix;

    // Now we need to render the linked list of meshes using this transform
    pD3DDevice->SetTransform( D3DTS_WORLD, &WorldMatrix );
    for ( pMeshContainer = pFrame->pMeshContainer;  pMeshContainer;
          pMeshContainer = pMeshContainer->pNextMeshContainer )
    {
         RenderMesh ( pMeshContainer );
    }

    // Process Sibling Frame with the same parent matrix
    if ( pFrame->pFrameSibling )
        RenderFrame (pFrame->pFrameSibling, pParentMatrix);

    // Move onto first child frame
    if ( pFrame->pFrameFirstChild )
        RenderFrame ( pFrame->pFrameFirstChild, &WorldMatrix);
}
```

In the above code, the RenderMesh() function is assumed to be a function that handles the actual drawing of the mesh. This can be a function that loops through each subset of the mesh, sets the relevant textures and material for that subset and then calls DrawSubset, or one that sets aside the mesh in some sort of render queue for additional sorting prior to the actual polygon drawing routines. If you choose the latter approach, just remember to store the world matrix for later access (and skip the SetTransform call since it is an unnecessary expense in that case).

To kick off the process above, we would call the function from our application render loop, passing in a pointer to our hierarchy root frame and NULL for the initial matrix pointer.

```
RenderFrame (pRootFrame, NULL);
```

Alternatively, we could pass in a standard world matrix (assuming the root frame matrix is set to identity by default – often a good idea) to move or orient the hierarchy in the world or with respect to some other parent if desired (like our car transporter example discussed earlier). Assuming our hierarchy was wrapped in some application defined object:

```
RenderFrame (object->pRootFrame, &object->world_matrix);
```

Also keep in mind our earlier discussion regarding intersection testing (ex. frustum culling). As mentioned, intersection testing is much more efficient when we use hierarchical spatial data structures due to the bounding volume relationship that ensues (parents completely enclose their children).

Let us look at dividing our rendering strategy into two passes. The first pass will update the world matrices for each frame. This would also be an ideal place to run any animation controllers if they were attached directly to nodes, but we will not worry about that for the moment. The second pass will handle the actual drawing calls, but will include simple visibility testing. Note that because we are separating the two passes, we will need to store the world matrix generated during the initial pass. Otherwise the render pass will not be able to properly transform the child meshes. We will assume in our example that we have derived our own frame type from D3DXFRAME called CMyFrame which adds a WorldMatrix D3DXMATRIX member for this purpose.

```
void Update ( LPD3DXFRAME pFrame, const D3DXMATRIX * pParentMatrix )
{
    // Cast to a pointer to our derived frame type
    CMyFrame *myFrame = (CMyFrame *)pFrame;

    // Could process local animations here…
    //(D3DX does things differently though as we will see)

    // If there is a parent matrix then combine this frame relative matrix with it
    // so we get the complete world transform for this frame.
    // Otherwise, this is the root frame, so this frame's matrix is
    // the world matrix for the frame -- so just copy it
    if ( pParentMatrix != NULL)
           D3DXMatrixMultiply(&myFrame->WorldMatrix, &myFrame->TransformationMatrix,
                              pParentMatrix);
    else
           myFrame->WorldMatrix = myFrame->TransformationMatrix;

     // Process Sibling Frame with the same parent matrix
     if ( myFrame->pFrameSibling )
          Update ( myFrame->pFrameSibling, pParentMatrix );

     // Move onto first child frame
     if ( myFrame->pFrameFirstChild )
           Update ( myFrame->pFrameFirstChild, &myFrame->WorldMatrix);
}
```

With our hierarchy now updated in world space, we can proceed to render it in a separate pass using the following function. Note that because the hierarchy has been updated for this render pass, all the frame structures now contain the actual world transformation matrices in their (newly added) WorldMatrix member. This function does not have to do any matrix concatenation; it just has to set the world matrix for any frame that contains meshes, and then render those meshes.

```
void Render ( LPD3DXFRAME pFrame, Camera  *pCamera )
{
     // Cast to a pointer to our derived frame type
     CMyFrame *myFrame = (CMyFrame *) pFrame;

     // If frame is not visible, neither are its children… just return
     if ( !pCamera->FrustumSphereTest( myFrame->center, myFrame->radius ))
            return;

     // Now we need to render the linked list of meshes using this transform
     pD3DDevice->SetTransform( D3DTS_WORLD, &myFrame->WorldMatrix );
     for ( pMeshContainer = myFrame->pMeshContainer;  pMeshContainer;
           pMeshContainer = pMeshContainer->pNextMeshContainer )
     {
           RenderMesh ( pMeshContainer );
     }

     // Process Sibling Frame with the same parent matrix
     if ( myFrame->pFrameSibling ) Render ( myFrame->pFrameSibling, pCamera );

     // Move onto first child frame
     if ( myFrame->pFrameFirstChild ) Render ( myFrame->pFrameFirstChild, pCamera );
}
```

In this code example, our visibility test is just about the simplest one possible. It is a very rough cull that considers partial intersections to be visible. We could instead implement a more robust system that accounts for cases where the bounding volume is fully outside, fully inside, or partially intersecting the frustum. When the volume is fully inside, no additional tests are needed for the children – they can simply be rendered directly without them having to do tests of their own. When the volume is partially inside, we can run additional tests with tighter fitting bounding volumes (perhaps an AABB) if desired. We will discuss Bounding Volume/Frustum tests again later in this course.

There are other optimizations that can be done with frustum culling as well. For example, given what we know about our parent-child relationship, we can further speed up our routine by tracking which frustum planes the parent tested against and letting the children know this during the traversal (a bit flag works nicely here). If the parent was completely inside the left frustum plane for example, so are its children and they will not need to run that test again -- only partial intersections need to be tested. This reduces the number of tests that need to be run at each node in the tree. Again, a parent being fully outside any single plane means all of its children are as well and traversal stops at that point as we saw above.

Another popular optimization is frame-to-frame coherency which adds even more value to the previous two optimizations. A simple implementation of this requires that we keep track of the last failed frustum plane tested between render calls (usually as a data member in our derived frame class). It is likely that on the next render pass, that frame will fail against the same frustum plane, so we can test it first. If the frame fails that same frustum plane test, we can immediately skip the remaining 5 (out of 6) frustum plane tests and return immediately, letting the caller know the frame is still not visible. We will be incorporating all of these concepts and more into our game engine design during the next course in this series, but you should feel free to try your hand at them sooner if you like. They really can make a big difference in engine performance.

Conclusion

This chapter has certainly been tough going, due mainly in part to the fact that the loading of data is rarely considered very engaging subject matter. We have not even learned anything that will give us instant visual gratification. However, as virtually all game engines arrange their data hierarchically, what we have learned in this chapter is a foundation upon which everything else we do will be built from this point on. Animated game characters for example (Chapter 11) will be represented as frame hierarchies, where each frame in the hierarchy represents an animation-capable joint of the character's skeleton.

As we have now learned how to load, construct, traverse, and render frame hierarchies, we now finally have the ability to load and represent entire scenes as independent objects that can be manipulated as needed.

We will use hierarchies to spatially subdivide the world to make efficient rendering possible as we will see later. In fact, frame hierarchies serve as a great introduction to the topic of scene graphs. Scene graphs are basically a more complex and fleshed out frame hierarchy implementation. We will be discussing them in module III of this course series when we leave our framework behind and migrate towards larger scale graphics engine technologies.

Simply put, you will find that we will be using frame hierarchies almost everywhere from now on. Indeed, we will begin with the very next chapter. In Chapter Ten, we will learn how to use the D3DX animation system to animate our scene hierarchies. The D3DX animation system allows us to perform some very complex tasks, from playing cut scenes using our in game graphics engine, to performing real time animation blending. Hierarchies play a pivotal role in these tasks, so you are now well equipped to move forward with confidence.

Workbook Nine:
Frame Hierarchies

Lab Project 9.1: Multi-Mesh Hierarchies

The focus of this lab project will be on loading and rendering multi-mesh frame hierarchies. Many of the topics discussed in the accompanying textbook will be demonstrated as we progress. For example, we will implement a class called CActor which encapsulates hierarchies and provide the application with an easy means to update and render them. We will also derive our own class from ID3DXAllocateHierarchy to facilitate the automated loading of multi-mesh X files within CActor. In addition, we will explore the relative nature of frame hierarchies by creating a simple animation class (called CAnimation). As discussed in the textbook, objects of this type can be attached to frames in the hierarchy and used to animate/update their matrices in a local fashion. While the demonstration application we create (Figure 9.1) may appear to be a simple mesh viewer, the car is actually a multi-mesh representation. This allows us to apply animations to the wheels separately from the rest of the hierarchy. Finally, the frames in our hierarchy will store pointers to meshes of type CTriMesh. As a result, we can continue to use the mesh resource management system we developed in the previous chapter and render our meshes using the class interface we are now familiar with.

Figure 9.1

This lab project will teach you how to do the following:

- Populate a frame hierarchy from an X file.
- Derive your own class from the ID3DXAllocateHierarchy interface.
- Populate the frame hierarchy from external references inside an IWF file.
- Animate individual frames in the hierarchy.
- Traverse and render a hierarchy.
- Represent meshes in the hierarchy using our own object type (CTriMesh).

Introducing CActor

To encapsulate hierarchies (loaded from an X file in this case), the CActor class will be implemented and employed. If multiple hierarchical X files need to be loaded (e.g., multiple game characters) then a separate CActor object will be instantiated to wrap the frame hierarchy and mesh data for each one. The term 'Actor' is commonly used in many game development SDKs to represent an animation capable object or hierarchy of objects that plays a part in the scene. This is not unlike the way an actor in Hollywood plays a part in a movie. While this is exactly the context in which we are using the name Actor, it should be made clear that our CActor class need not just represent the frame hierarchy for a single multi-mesh model (e.g., a character or an automobile) but may represent an entire scene hierarchy loaded from a single X file.

The CActor class will wrap the call to the D3DXLoadMeshHierarchyFromX function with its own method called LoadActorFromX. This is the function that our application will call to load the X file into the actor. When it returns control back to the application, the hierarchy and mesh data that was contained in the file will be loaded and the CActor will contain a D3DXFRAME derived pointer to the root frame of the hierarchy. The CActor class will also expose functions to traverse the hierarchy and update the frame matrices to generate the absolute matrices for each frame prior to rendering its meshes. Additionally, the interface will expose functions to render the hierarchy and its contained meshes.

In the textbook we learned that in order to use the D3DXLoadMeshHierarchyFromX function, we must derive a class from the ID3DXAllocateHierarchy interface. You will recall that the four member functions of this interface are used as callbacks by the D3DX loading function to allow the application to allocate the memory for the hierarchy frames and mesh containers in the appropriate way. Our ID3DXAllocateHierarchy derived class will be a standalone class that is instantiated and used by the CActor object to pass into the D3DXLoadMeshHierarchyFromX function. The CreateMeshContainer callback function of the CAllocateHierarchy class that we derive will also need access to the CActor object which is currently using it. Therefore, when we derive our class, we will also add a pointer to a CActor object that will be set by its constructor.

The reason the CAllocateHierarchy::CreateMeshContainer function needs to know about the CActor object using it harkens back to the previous lesson where we wanted to have texture and material management for all meshes (contained in all actors) handled at the scene level. You will recall that we implemented a callback mechanism for our CTriMesh class so that when it loaded data from an X file, the returned texture and material buffer was passed off to the scene (via the scene callback functions that had been registered with the mesh) to add the textures and materials to the scene's texture and material arrays. With meshes in non-managed mode, this meant that multiple CTriMeshes in the scene shared the same resources and needed to be rendered by the scene itself. This provided better batching potential. Even in non-managed mode (where the CTriMesh stored its own texture and material information) the textures were still loaded and stored in the scene object to minimize redundancy when multiple managed meshes used the same texture resources. We very much wish to use our CTriMesh class to represent the various meshes stored in our D3DX frame hierarchies so that we can work with an interface that we are comfortable with and continue to benefit from its cloning and optimization functions. Furthermore, we definitely want our meshes to benefit from the managed and non-managed mode resource management systems implemented by that class.

Unlike using our CTriMesh class in isolation, where creation is basically performed by the D3DXLoadMeshFromX function, our meshes will now need to be created manually inside the CreateMeshContainer function of the CAllocator object. Recall that this function is called for each mesh in the hierarchy by D3DX and it is passed the ID3DXMesh interface and the materials the mesh uses. Therefore, we do not wish to use the CTriMesh::LoadMeshFromX function (as we did in the previous lesson) to create our CTriMesh data because the mesh data has already been loaded by D3DX. Instead, our CTriMesh instances will need to be created inside the CreateMeshContainer function and attach the ID3DXMesh data that we are passed. This presents a small problem because previously our application would instantiate a CTriMesh, register the material callbacks, and then call the CTriMesh::LoadMeshFromX function which triggered these callbacks. But now that the CTriMesh::LoadMeshFromX function will no longer be used, our CTriMesh is not created until the hierarchy is being loaded. Therefore, the application cannot register callbacks with meshes that do not yet exist prior to the D3DXLoadMeshHierarchyFromX call made by CActor. The simple solution is to duplicate some of this callback functionality inside CActor. This will allow the application to register the same three callback functions with the CActor instead of CTriMesh. When the CreateMeshContainer function is called, because the CAllocateHierarchy object has a pointer stored for the CActor for which it is being used, it can access the actor's function callback array and perform the same task (handing resources for each mesh off to the scene).

The basic task of the CAllocateHierarchy::CreateMeshContainer function will be as follows:

- Allocate a new CTriMesh object and attach it to the ID3DXMesh that it is passed by D3DX
- If the CActor is in managed mode, allocate space in the CTriMesh's internal attribute table to store the texture pointer and material for each subset that we are passed by D3DX. Copy the material data into the CTriMesh's internal attribute table and use the CActor's registered texture callback function to load the texture and return a texture pointer. These texture pointers are also stored inside CTriMesh's attribute table.
- If the CActor is in non-managed mode then the function will simply send the passed material data to the actor's registered attribute callback function which will store the texture and material data for each subset at the scene level. This function will return a new global attribute ID for each subset in the mesh which the function can use to lock the CTriMesh's index buffer and remap its attribute buffer.
- Store the newly created CTriMesh in the mesh container and return it to D3DX so that it can be attached to the frame hierarchy.

All of the above steps should seem very familiar. This code was seen in the previous chapter inside the CTriMesh::LoadMeshFromX function. All we have done is essentially duplicated it inside the CreateMeshContainer method of the CAllocateHierarchy object. This same code still exists in the CTriMesh::LoadMeshFromX function so that it can continue to be used as a standalone mesh object that does not have to be used with CActor or frame hierarchies. However, when the CTriMesh is part of the CActor's internal hierarchy, their own callback system will not be used and the CActor will manage the callbacks instead.

It is also worth noting that our CActor class can still be used to load an X file even if that X file does not contain a frame hierarchy and contains only a single mesh. In such a case, the CActor's hierarchy will consist of just a single frame (the root) with the mesh attached as an immediate child

object. Single mesh CActors can be rendered and updated in exactly the same way, so the difference is invisible to the application using the CActor class.

Application Design

An understanding of the relationships between our various classes is vital. So let us break down the responsibilities for each object:

CScene

As we have come to expect, the scene object handles the loading of scene data and the rendering of that data at a high level. The application uses the CScene::LoadSceneFromX or CScene::LoadSceneFromIWF functions to load the data and set it up correctly. The scene loading methods have now been modified to be aware of actors.

The CScene::LoadSceneFromX function will now allocate a new CActor and register any resource callback functions with the actor before calling the LoadActorFromX function to populate the frame hierarchy with X file data. Each call to this function made by the application will load a new X file and will create a new CActor in the scene's CActor array. In our application, this function is called once to load the automobile representation from the X file.

The CScene::LoadSceneFromIWF (and all its support functions) are largely unchanged. In the previous chapter, it searched for external references stored inside the IWF file and used their names to load an X file into a CTriMesh. Now, this same function will allocate a new CActor when an external reference is found instead. It will then register the scene's resource callbacks with the newly created actor before calling the CActor::LoadActorFromX function to load the X file data.

All the other scene methods that are part of the framework still exist (AnimateObject, Render, etc.) and will be changed slightly to work with actors.

CActor

The CActor class encapsulates the loading, updating, and rendering of a hierarchy.

Its loading function CActor::LoadActorFromX will instantiate a temporary CAllocateHierarchy object and pass it into the D3DXLoadMeshHierarchyFromX function. When this function returns, the CActor's root frame pointer will point to the root frame of the newly loaded frame hierarchy. Any meshes attached to frames in the hierarchy will be of type CTriMesh. The scene will call this function to load any X files (either directly or via external references in an IWF file).

The CActor::RegisterCallback function is virtually identical to its cousin in the CTriMesh class. It allows the application to register callback function pointers. We covered the code to the scene's three resource callback functions in the previous workbook and these are unchanged. This CActor method simply stores the passed function pointers in an array so that they are accessible from the CAllocateHierarchy::CreateMeshContainer function (via its CActor pointer) when the hierarchy is being loaded.

This class will also have a method called DrawActorSubset which will be called by the scene to render **all** CTriMesh subsets in its hierarchy with a matching attribute ID. This function will traverse the hierarchy calling CTriMesh::DrawSubset for each of its contained meshes.

CTriMesh

This class is unchanged from our previous workbook. However, now objects of this type will be stored in the frame hierarchies of the actors to which they belong. In our application, we will be using all the meshes in non-managed mode, so the scene class will be responsible for setting the states before calling CActor::DrawSubset. This allows us to benefit from greater batching potential using global attribute IDs across all CTriMeshes in all CActors currently being used by the scene.

Although not demonstrated in this demo, it is also perfectly legal to use the CActor in managed mode. This is actually the default case when the attribute callback has not been registered with the CActor. When this is the case, each CTriMesh in the frame hierarchy will internally store the material and texture pointer used by the mesh for each of its subsets.

CAllocateHierarchy

The CAllocateHierarchy object is also introduced in this application. It is derived from the ID3DXAllocateHierarchy interface and is instantiated by CActor and passed into the D3DXLoadMeshHierarchyFromX function.

This object will contain four callback functions (CreateFrame, CreateMeshContainer, DestroyFrame, and DestroyMeshContainer) which will be called by the D3DX loading function to allocate the memory for the various hierarchy components as the hierarchy is being constructed. This allows us to customize the mesh and frame data that we are passed. For example, the CreateMeshContainer method provides us the flexibility to store the passed mesh data in a CTriMesh and add that to the mesh container instead. It also allows us to clone the mesh data into another format or optimize its vertex and index data.

Our CAllocateHierarchy class will also have a single member variable: a pointer to a CActor. When the CAllocateHierarchy object is instantiated by the CActor, the pointer to the actor is passed into the constructor and stored in the object. It will be used in the CreateMeshContainer function to access the resource callback functions that have been registered with the actor. This will allow the array of material and texture data that we are passed by D3DX for a given mesh to be registered and stored at the scene level. It will also allow for the re-mapping of each mesh's attribute buffer into global attribute IDs.

CObject

The CObject class as been used in nearly all of our previous lab projects to represent meshes in the scene. This was necessary because a mesh itself does not store a world matrix. For example, in the last workbook, this object coupled together a CTriMesh with its assigned world matrix. CObject will now have a new member pointer added: a pointer to a CActor. Usually, only one of the pointers will be active at once (either the CActor pointer or the CTriMesh pointer) depending on whether the object represents just a single CTriMesh or a CActor. Although making these two pointers a union would seem the logical choice, there may be times when we might wish the CObject to represent

both an actor and a separate mesh that both use the same world matrix. So the design decision was made to allow both pointers to exist simultaneously.

When the application uses the CScene::LoadSceneFromX function, a new CObject will be added to the scene. That object will contain a matrix and an actor. When the application calls CScene::LoadSceneFromIWF, several CObjects will typically be added to the scene. Meshes stored internally in the IWF (such as brushes from GILES™) will be loaded into a CTriMesh and attached to a new CObject and added to the scene's CObject array. Any external references stored in the IWF file will be loaded into a CActor, attached to a new CObject, and added to the scene CObject array. Therefore, when the scene is loaded from an IWF file, multiple CObjects will be added to the scene. Some objects in this array may contain actors (i.e., mesh hierarchies) while other objects may contain only simple CTriMesh pointers. Our scene rendering function will need to be able to render both types of CObject.

We know from previous lessons that the matrix stored in the CObject represents the world matrix that will be sent to the device before rendering the contained mesh. This behavior is still true when the CObject is being used to represent only a single mesh. However, when a CActor is encountered, this matrix is not used in quite the same way. Instead, this matrix will be used to position and orient the root frame of the hierarchy. The scene object will not simply set this matrix as the world matrix on the device since we know that would not work -- the actor might contain multiple meshes which all need to have their absolute world matrices generated and set separately before they are rendered.

The scene object will instead call the CActor::SetWorldMatrix method, passing in the matrix stored in CObject. The CActor will use this passed matrix to combine with the root frame matrix and traverse the tree, generating all absolute world matrices for each frame. As we know from our textbook discussions, combining the CObject matrix with the root frame matrix will affect all the world matrices that are generated below the root frame in the hierarchy. Once all absolute matrices in the hierarchy have been updated, we have all the matrices we need to render any mesh in the hierarchy. When we render a mesh, we simply use the absolute matrix stored in its owner frame as the world matrix that we set on the device. This is done in a separate rendering pass through the hierarchy.

From the application's perspective, things are no different with respect to moving objects around in our scene. When we pass the CObject's matrix into CActor::SetWorldMatrix, we are applying a transformation to the root frame which filters down to all other frames in the hierarchy. Therefore, we can still move the object around in our scene using only the CObject matrix.

The following diagram shows the relationships that exist between the various classes in our application. Notice that the scene stores a list of objects, each object stores an actor and each actor stores the root frame to an entire frame and mesh hierarchy.

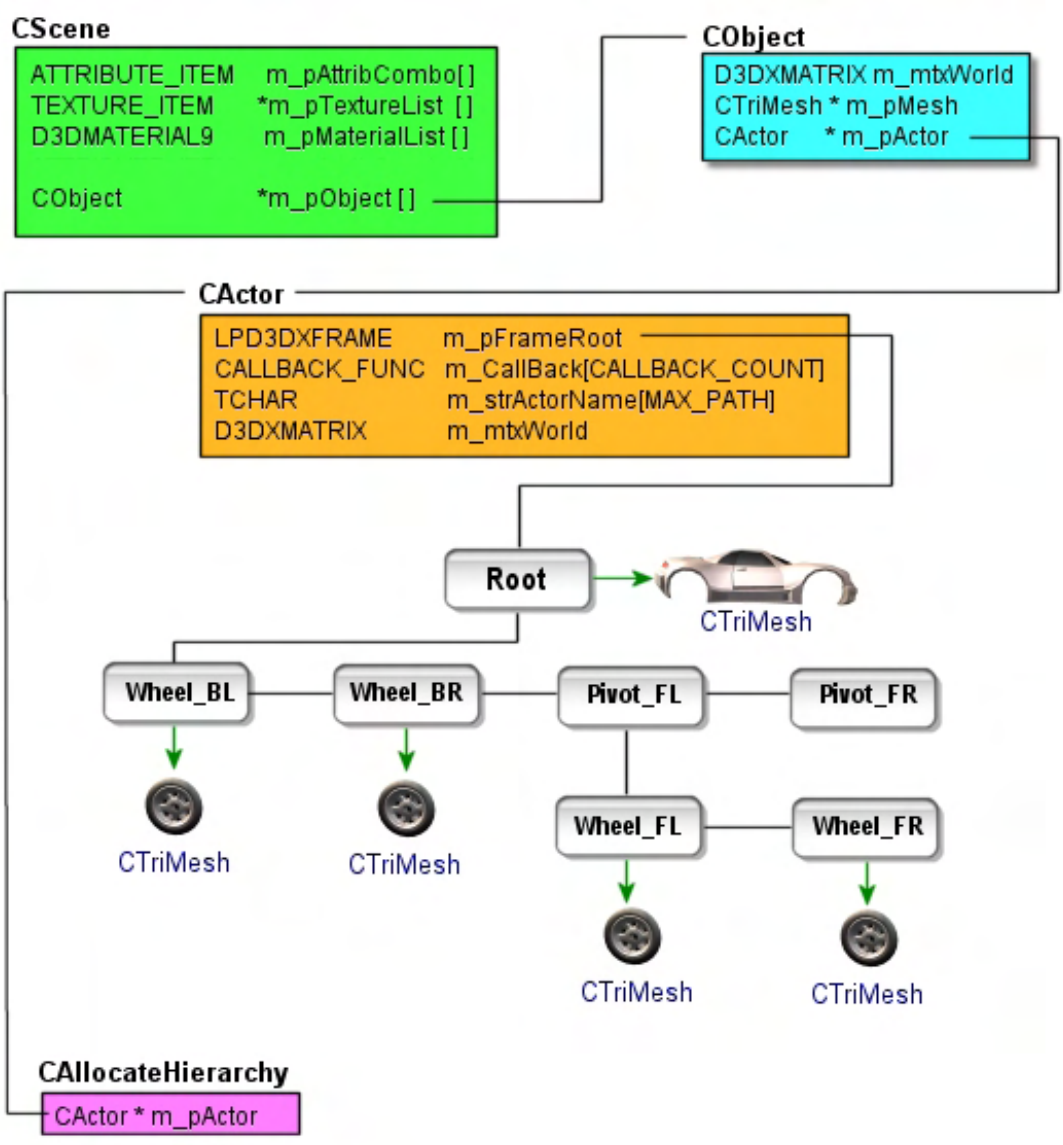


Figure 9.2

The CAllocateHierarchy object in Figure 9.2 is only instantiated temporarily (on the stack) by CActor for the call to D3DXLoadMeshHierarchyFromX.

Notice how the CActor itself also stores its own matrix. In fact, when we call CActor::SetWorldMatrix and pass in the CObject's matrix, this matrix is cached internally by the actor. The reason we do this is that many of the methods exposed by CActor (Draw, UpdateFrames, SetWorldMatrix, etc.) would usually require a total traversal of the hierarchy and a rebuilding of the absolute frame matrices stored in each frame. You may for example, issue several function calls to the actor which would cause the hierarchy to be traversed each time. To rid ourselves of unnecessary traversals, each function (such as SetWorldMatrix) will also except a Boolean parameter indicating whether we wish the function to update the matrices in the hierarchy immediately or whether we simply wish to store the state such that it can be used to update the hierarchy later via another function. This allows us to set the actor world matrix, apply some animation to the relative frame matrices, and then later call

CActor::UpdateFrameMatrices to traverse the hierarchy and rebuild the absolute matrices using the world matrix that was set in a previous call . Using this technique, the first two function calls would not update the hierarchy, but would simply set states. The final call would use these states to update the hierarchy in a single pass using those caches states. Without this mechanism, every call that alters the hierarchy would cause a hierarchy traversal and a rebuild of all absolute frame matrices.

Another thing that may seem strange in the above diagram is that our automobile frame hierarchy looks rather different from the diagrams we looked at in the textbook. There is another level of frames (called *pivots*) between the root and the front wheels. The reason these pivots are necessary is clear when we establish what our application will do. Every time the scene is rendered, the wheel frames of the car will be rotated around their local X axis. We will discuss the class that performs this task in a moment. The process is basically one of building an X axis rotation matrix that rotates some small number of degrees and multiplying it with the relative frame matrices for each wheel. This accumulative step will rotate the wheels a little more around their **local** X axis every cycle of the game loop. This will cause them to spin around like the wheels on a real automobile. The local X axis of each wheel can be perceived as sticking out of the page directly at you if you were viewing the wheel face on (Figure 9.3).

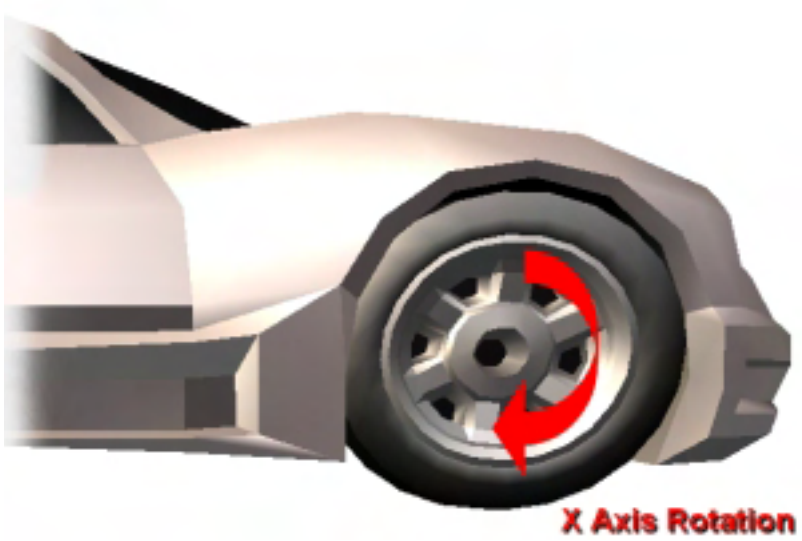


Figure 9.3

The red arrow shows the rotation around the X axis (which we cannot see in this diagram because we would be looking directly down it). It should be clear why rotating the wheel frame matrix X axis is the desired choice. The textbook taught us that applying rotations to the relative matrix of a frame causes a local rotation. Therefore, however much we rotate the root frame and change the orientation of each wheel in the world, each wheel will always rotate around the correct axis: the local X axis.

The problem we have is when we introduce the need for additional local rotations for that wheel frame. In our application, we also wish to give the user the ability to press the comma and period keys to steer the wheels left or right. If we look at Figure 9.4 we can see the local Y axis prior to any local rotations. It is clear that this is the axis we would wish to use for left/right wheel rotation.

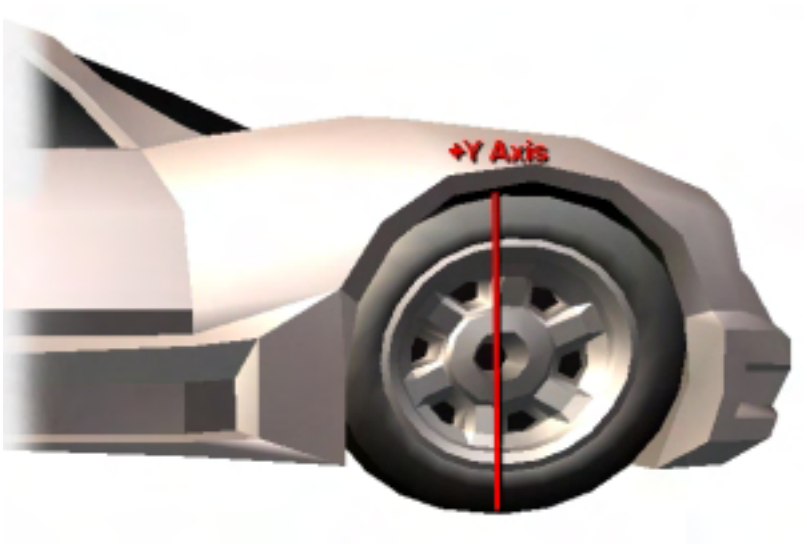


Figure 9.4

In Figure 9.4 we see the local Y axis of the wheel frame prior to any rotation. Just as the local X axis can be thought of as the wheel's right vector, this axis describes the wheels up vector. So far it all looks very promising.

So it would seem at first that our task is a simple one. For every update, we build an X axis rotation matrix and combine it with the wheel matrix causing the wheel to revolve. Then, when the user presses a comma or period key, we simply build a Y axis rotation matrix (with either a negative or positive degree value for left and right respectively) and apply that to the frame matrix of the wheel as well. Right?

Actually, no! The problem is that when we combine matrices with a relative frame matrix we are dealing with its local coordinate system. Imagine for example that the wheel has been rotated 45 degrees about its X axis and we then, based on user input, wanted to apply a Y rotation to make the wheel turn. The problem is, at this point, the entire local coordinate system has been rotated about the local X axis. So the local Y axis (the wheel up vector) is no longer pointing straight up relative to the car body. Instead it is oriented at an angle of 45 degrees around the wheel's local X axis (Figure 9.5). It is clear that applying a Y rotation to this matrix would not have the correct effect at all.

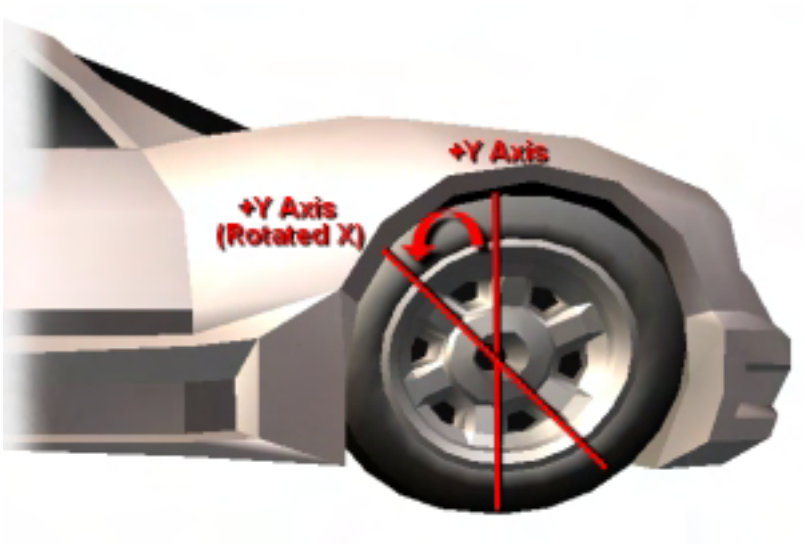


Figure 9.5

So, it seems we cannot use the frame matrix for both X *and* Y rotation. Maybe we could apply the Y rotation to the parent frame instead? No, that would not work either because the parent frame in this case would be the root frame which represents a position at the center of the automobile. While the orientation of the root frame's Y axis absolutely is the axis we need to rotate about, the wheel is offset in this frame of reference from that origin. We would not want the wheel to rotate about the center of the automobile instead of around its own center point. Applying a rotation to the parent would actually rotate the whole car about its Y axis, so that definitely is not the way to go either. We need these wheels to rotate about their own coordinate origins, but using the automobile up vector.

As it happens, there is a very simple way to do what we need and it is commonly used by artists to help game programmers with such tasks. The use of *dummy frames* in the hierarchy, which to the uninitiated may seem to prove no purpose, solves the problem. Two dummy frames exist in our automobile representation and they are positioned above the two front wheels in the hierarchy. Let us examine their purpose.

Figure 9.6 shows the position of the two dummy frames (Pivot_FL and Pivot_FR) in the hierarchy. These frames are positioned in the root frame of reference such that they describe a position at the center of the wheels. They have no default orientation (i.e., identity). In other words, these pivot frames describe the position of the wheels themselves (like the diagrams in the textbook). However, each pivot frame has a child wheel frame. These wheel frames also initially start out with an identity orientation but (and this is the important point) they also have a *zero translation vector*. Since they are defined as children in the pivot frame of reference, but with zero offset, this means that for each front wheel we have two frames on top of each other sharing the same space. To demonstrate this important point, if we were to also assign a wheel mesh to Pivot_FL and render this hierarchy, the mesh at Pivot_FL and the mesh at Wheel_FL would be in the exact same position.

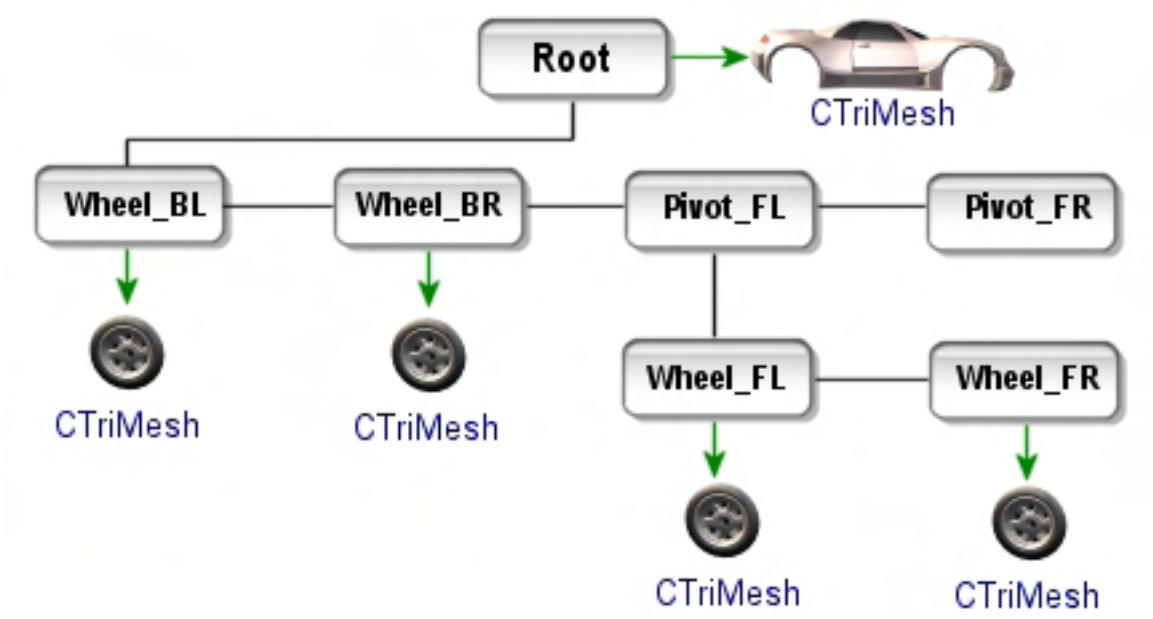


Figure 9.6

The usefulness of these seemingly redundant frames becomes clear when we consider the very important fact that Wheel_FL is defined in the Pivot_FL frame of reference.

If we apply a Y axis rotation to the Pivot_FL frame, we know that any child frames will inherit this rotation. We know for example that when we apply a rotation to the root frame, the four wheel frames will rotate about the root frame Y axis at a radius consistent with their specified offsets. In this case however, the Wheel_FL frame has not been offset from its parent frame. Therefore, we can think of the Pivot_FL frame's Y axis as also passing through the origin of the wheel frame's local coordinate system. Applying a rotation to the pivot will cause the wheel frame (and its mesh) to rotate about its own center point but also around the parent axis. The great thing is that because we have applied the Y axis rotation to the parent frame, we are free to rotate the wheel frame about its local X axis without any fear that its orientation will interfere with (or be corrupted by) other rotation operations.

Our task is simple then. During every update we will apply an X axis rotation to each of the wheel frame matrices, and when the user presses either the comma or period key, we will apply a Y axis rotation to the pivot frames. Problem solved.

CAnimation

The last new class we will introduce in this lesson is called CAnimation. This is a very simple object that will be used to explore hierarchical animation and prepare you for the following chapter where we will explore the DirectX animation controller in detail. The job of this object will be to apply rotations to frames in the hierarchy. There will be one CAnimation object created for each frame that needs to be animated.

The scene will create six animation objects (one for each of the wheels and two more for the pivot frames) and will use the CAnimation::Attach method to attach frame matrices in the hierarchy to the individual animation objects. The CAnimation object simply stores this matrix internally so that it

knows which matrix to update when the scene object calls its RotationX, RotationY and RotationZ methods. It is these methods that build a rotation matrix and multiply it with its associated frame matrix. As you can probably imagine, the code to this class is extremely small.

> **Note**: Using this naming convention, an animation is assumed to be either an object or dataset that animates a single frame in the hierarchy. In our example X file, the car has six frames that need to be animated. Therefore, our demo contains six animations.

Source Code Walkthrough – CActor

With a high level discussion of this lab project behind us, we will now examine the code sections that we are not yet familiar with. We will start by looking at the CActor class, which is where most of the stuff that is new to us will be hiding. The class definition is contained in CActor.h and is shown below in its entirety. This should give you a feel for the interface it exposes to the application and the member variables it contains. This will be followed with a description of its member variables.

```
class CActor
{
public:
    enum CALLBACK_TYPE { CALLBACK_TEXTURE=0,
                         CALLBACK_EFFECT = 1,
                         CALLBACK_ATTRIBUTEID = 2,
                         CALLBACK_COUNT = 3 };
    // Constructor
             CActor( );
    virtual ~CActor( );

    bool      RegisterCallback   ( CALLBACK_TYPE Type, LPVOID pFunction, LPVOID pContext );
    HRESULT   LoadActorFromX     ( LPCTSTR FileName, ULONG Options,
                                   LPDIRECT3DDEVICE9 pD3DDevice );
    void      Release            ( );
    void      SetWorldMatrix     ( const D3DXMATRIX * mtxWorld = NULL,
                                          bool UpdateFrames = false );
    void      DrawActor          (  );
    void      DrawActorSubset    ( ULONG AttributeID );

    // Accessor functions
    LPCTSTR       GetActorName      ( ) const;
    CALLBACK_FUNC GetCallback       ( CALLBACK_TYPE Type ) const;
    ULONG         GetOptions        ( ) const;
    LPD3DXFRAME   GetFrameByName    ( LPCTSTR strName, LPD3DXFRAME pFrame = NULL ) const;

    // Private Functions
    private:
    void    DrawFrame           ( LPD3DXFRAME pFrame, long AttributeID = -1 );
    void    DrawMeshContainer   ( LPD3DXMESHCONTAINER pMeshContainer,
                                  LPD3DXFRAME pFrame, long AttributeID = -1 );
    void    UpdateFrameMatrices ( LPD3DXFRAME pFrame, const D3DXMATRIX * pParentMatrix );

    // Member Variables for This Class
    LPD3DXFRAME                  m_pFrameRoot;
    LPDIRECT3DDEVICE9            m_pD3DDevice;
    CALLBACK_FUNC                m_CallBack[CALLBACK_COUNT];
    TCHAR                        m_strActorName[MAX_PATH];
    D3DXMATRIX                   m_mtxWorld;
    ULONG                        m_nOptions;
};
```

LPD3DXFRAME m_pFrameRoot

When the actor's hierarchy is constructed in the CActor::LoadActorFromX method, this pointer is passed into the D3DXLoadMeshHierarchyFromX function and used to store the address of the root frame that is returned. This single pointer is the actor object's doorway to the hierarchy and the means by which it begins hierarchy traversal.

Below, you are reminded of the layout of the D3DXFRAME structure:

```
typedef struct _D3DXFRAME
{
    LPSTR Name;
    D3DXMATRIX TransformationMatrix;
    LPD3DXMESHCONTAINER pMeshContainer;
    struct _D3DXFRAME *pFrameSibling;
    struct _D3DXFRAME *pFrameFirstChild;
} D3DXFRAME, *LPD3DXFRAME
```

Because we need to store additional information in our frame structure, such as the absolute world matrix of a frame (and not just the relative matrix provided by the vanilla D3DX version shown above), we will derive our own structure from D3DXFRAME to contain this additional matrix information. The frame structure used throughout our actor's hierarchy is shown below.

```
struct D3DXFRAME_MATRIX : public D3DXFRAME
{
    D3DXMATRIX  mtxCombined;    // Combined matrix for this frame.
};
```

In the textbook, we also covered the D3DXMESHCONTAINER structure that is used throughout the hierarchy to contain mesh data for a frame. You are reminded of this structure again below. It should be very clear to you what each member of this structure is used for after our discussion in the textbook.

```
typedef struct _D3DXMESHCONTAINER
{
    LPSTR Name;
    D3DXMESHDATA MeshData;
    LPD3DXMATERIAL pMaterials;
    LPD3DXEFFECTINSTANCE pEffects;
    DWORD NumMaterials;
    DWORD *pAdjacency;
    LPD3DXSKININFO pSkinInfo;
    struct _D3DXMESHCONTAINER *pNextMeshContainer;
} D3DXMESHCONTAINER, *LPD3DXMESHCONTAINER
```

We will also wish to derive from this structure because we want to represent our meshes in the hierarchy using the CTriMesh object we created in the previous workbook. You will recall that the D3DXMESHDATA member of this structure would normally contain the ID3DXMesh interface for the mesh being stored here. The structure that we will use is derived from this one, with a single extra member added: a CTriMesh pointer:

```
struct D3DXMESHCONTAINER_DERIVED : public D3DXMESHCONTAINER
{
    CTriMesh * pMesh;          // Our wrapper mesh.
};
```

As you can see, our mesh container now contains a pointer to a CTriMesh. The CAllocateHierarchy::CreateMeshContainer function will take the passed ID3DXMesh, create a CTriMesh from it and attach it to the mesh container, instead of storing the ID3DXMesh pointer directly in the mesh container's D3DXMESHDATA member.

> **Note:** In our application we also store the CTriMesh's underlying ID3DXMesh interface pointer in the mesh container's D3DXMESHDATA member. While there might seem little point in doing this, D3DX provides many helper functions that calculate things like the bounding volume of a frame. These functions expect the mesh data to be stored in the D3DXMESHDATA member of the frame as they have no knowledge that we are now using our own CTriMesh object to store and render mesh data. Without us also storing the mesh interface used by the CTriMesh in this member, these functions will not work correctly.

The remaining member variables of the CActor class will now be discussed.

LPDIRECT3DDEVICE m_pD3DDevice

When the application calls the CActor::LoadActorFromX function it must also pass a pointer to the Direct3D device which will be the owner of any meshes in that hierarchy. The CActor passes this device pointer into the D3DXLoadMeshHierarchyFromX function and also stores the address in this member variable. It will need access to this pointer during rendering traversals of the hierarchy so that the absolute world matrices of each frame can be set as the device world matrix prior to rendering any of its attached mesh containers.

CALLBACK_FUNC m_CallBack[CALLBACK_COUNT]

As with the CTriMesh class, the actor has a static array that can be used to store pointers to resource callback functions. The CALLBACK_TYPE enumeration is also familiar to us because the CTriMesh class used this same enumeration within its namespace. The members of this enumeration describe the three possible callback functions which can be registered with the actor. The value of each member in this enumeration also describes the index in the callback array where the matching function pointer will be stored. For example, the CALLBACK_ATTRIBUTEID function pointer will be stored in m_CallBack[2].

TCHAR m_strActorName[MAX_PATH]

This string is where the name of the actor will be stored. This name is the name of the X file passed into the LoadActorFromX method. The actor's name will often be retrieved by the scene object to make sure that the same hierarchical X file does not get loaded more than once. When an external reference is found in an IWF file for example, a search of all the currently loaded actor names will be performed to make sure it does not already exist. If it does, then we will not create a new actor, but will simply point the new CObject's CActor pointer at the one that already exists. When we do this we are referencing the actor with multiple CObjects.

D3DXMATRIX m_mtxWorld

This is where the CActor will cache a copy of the matrix passed into its SetWorldMatrix function. This is the matrix that is used to perform updates to the hierarchy. This matrix will contain the world space position and orientation of the actor.

ULONG m_nOptions

This member will contain a combination of D3DXMESH creation flags used to create the meshes in the hierarchy. The application will pass these flags into the LoadActorFromX function where they will be stored and used later in the CAllocateHierarchy::CreateMeshContainer function to create the ID3DXMesh data in the application requested format. This is another reason why the CAllocateHierarchy object will need access to the CActor which is using it.

CActor::CActor()

The CActor constructor takes no parameters and initializes the object's member variables to zero. Its internal world matrix is set to an identity matrix, essentially describing this actor as currently existing at the origin of world space. The callback array is also initialized to zero.

```
CActor::CActor()
{
    // Reset / Clear all required values
    m_pFrameRoot      = NULL;
    m_pD3DDevice      = NULL;
    m_nOptions        = 0;

    ZeroMemory( m_strActorName, MAX_PATH * sizeof(TCHAR) );
    D3DXMatrixIdentity( &m_mtxWorld );

    // Clear structures
    for ( ULONG i = 0; i < CALLBACK_COUNT; ++i )
       ZeroMemory( &m_CallBack[i], sizeof(CALLBACK_FUNC) );
}
```

CActor::~CActor()

As with many of our classes, the destructor does not actually perform the cleanup code itself but rather calls the object's Release function to perform this task. This is beneficial, as the Release function can be used to wipe the information from a CActor without destroying it. This would allow us to use the same CActor object to load another X file once we are finished with the data it currently contains. As the Release function is going to have to be written anyway, let us not duplicate this same cleanup code in the destructor. So we simply call it:

```
CActor::~CActor( )
{
    // Release the actor objects
    Release();
}
```

CActor::Release()

A couple of items are quite interesting about this cleanup code. Firstly, notice the use of the D3DXFrameDestroy function to destroy the actor's entire frame and mesh hierarchy. This function was discussed in the textbook. We pass it an ID3DXAllocateHierarchy derived object and the root frame of the hierarchy to be destroyed. D3DX will then step through each frame in the hierarchy and call the DestroyFrame and DestroyMeshContainer callback methods of the passed allocator object. We will see that it is these callback functions that we must implement to perform the actual removal of frame and mesh data from memory. We will cover the CAllocateHierarchy class next.

So, this function instantiates a CAllocateHierarchy object and passes it into the global D3DXFrameDestroy function. The instance of the allocation object used to destroy the hierarchy does not need to be the same one used to create it, which is why we can create a temporary allocation object on the stack in this way. Notice the passing of the **this** pointer into the constructor, so that the allocation object can store a pointer to the actor which is using it.

```
void CActor::Release()
{
    CAllocateHierarchy Allocator( this );

    // Release objects (notice the specific method for releasing the root frame)
    if ( m_pFrameRoot      ) D3DXFrameDestroy( m_pFrameRoot, &Allocator );
    if ( m_pD3DDevice      ) m_pD3DDevice->Release();

    // Reset / Clear all required values
    m_pFrameRoot      = NULL;
    m_pD3DDevice      = NULL;
    m_nOptions        = 0;

    ZeroMemory( m_strActorName, MAX_PATH * sizeof(TCHAR) );
    D3DXMatrixIdentity( &m_mtxWorld );

    // Callbacks NOT cleared here!!!
}
```

When the D3DXFrameDestroy function returns, the entire frame hierarchy and all its meshes will be removed from memory. The Release function then relinquishes its claim on the device and sets its member variables to their initial values.

Notice however that the Release function does not clear the actor's resource function callback array. This is very convenient when you wish to re-use the object. Usually, even if you want to erase the actor's data and reuse it, the same callback functions can be used to load textures and materials. This allows us to release the actor's data without have to re-register the same callback functions. As the callback array is a statically allocated array, it will naturally be destroyed when the object is destroyed.

CActor::RegisterCallback

This function is called by the application to register any of three types of resource callback functions prior to calling the LoadActorFromX method. The passed function pointers and their associated contexts (in our demo, a CScene pointer) are simply stored in the callback array. These callback functions

essentially describe whether the actor is in managed or non-managed mode. If the function callbacks CALLBACK_TEXTURE or CALLBACK_EFFECT are being used, then all the meshes in the hierarchy will be created in managed mode. We are currently not using the CALLBACK_EFFECT callback until Module III when we cover effect files. If the CALLBACK_ATRIBUTEID callback is registered, all meshes in the hierarchy will be created in non-managed mode.

```
bool CActor::RegisterCallback( CALLBACK_TYPE Type, LPVOID pFunction, LPVOID pContext )
{
    // Validate Parameters
    if ( Type > CALLBACK_COUNT ) return false;

    // Store function pointer and context
    m_CallBack[ Type ].pFunction = pFunction;
    m_CallBack[ Type ].pContext  = pContext;

    // Success!!
    return true;
}
```

These function pointers are not actually used directly by CActor, but are accessed by the CAllocateHierarchy object each time a mesh container is to be created. If the CALLBACK_TEXTURE function pointer exists (m_Callback[0]), then the texture filenames for each material in the mesh being created will be passed to this callback. The callback is responsible for creating the texture and returning a pointer back to the CAllocateHierarchy::CreateMeshContainer function. The texture pointer and the material will be stored inside the CTriMesh, which will then manage its own state changes and have the ability to render itself in a self-contained manner.

If the CALLBACK_ATTRIBUTEID function has been registered (m_CallBack[2]) then the CAllocateHierarchy::CreateMeshContainer function will create all CTriMeshes in the hierarchy in non-managed mode. The texture and material information passed to the CreateMeshContainer function by D3DX will simply be dispatched to this callback. The scene will create and store the textures and materials in its own arrays and return a global attribute ID back to the CreateMeshContainer function for that texture and material combination. The CreateMeshContainer function will then lock the attribute buffer of the CTriMesh and re-map its attribute IDs to this new value. When in this mode, the scene object will be responsible for the state management and subset rendering of each actor's meshes.

CActor::LoadActorFromX

You would be forgiven for thinking that it is in the CActor::LoadActorFromX function that all the hard work happens, and therefore it is expected to be quite large in size. But as you can see, this function is actually very small since it uses the D3DXLoadMeshHierarchyFromX function to handle most of the heavy lifting. Indeed, it is in the CAllocateHierarchy::CreateMeshContainer function that most of the loading code is situated. We will take a look at that class next.

This function should be called by the application only after the resource callback functions have been registered. It will be passed the name of the X file that needs to be loaded along with any mesh creation flags. These mesh creation flags are a combination of D3DXMESH options, which we have used for mesh creation throughout the last lesson. They describe the resource pools we would like the mesh index

and vertex buffers created in. The function should also be passed a pointer to the Direct3D device which will own the meshes.

The first thing the function does is instantiate an object of type CAllocateHierarchy. The **this** pointer is passed into the constructor so that its CreateMeshContainer callback function will be able to access the actor's resource callback functions and the mesh creation flags for mesh container creation.

The passed device pointer is stored in the actor's member variable and the reference count incremented. The mesh creation options are also stored by the actor.

Next we call the function that kick starts the whole loading process, D3DLoadMeshHierarchyFromX.

```
HRESULT CActor::LoadActorFromX(LPCTSTR FileName,ULONG Options,LPDIRECT3DDEVICE9 pD3DDevice)
{
    HRESULT hRet;
    CAllocateHierarchy Allocator( this );

    // Validate parameters
    if ( !FileName || !pD3DDevice ) return D3DERR_INVALIDCALL;

    // Store the D3D Device here
    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();

    // Store options
    m_nOptions = Options;

    // Load the mesh heirarchy
    hRet = D3DXLoadMeshHierarchyFromX( FileName, Options, pD3DDevice, &Allocator,
                                       NULL, &m_pFrameRoot, NULL );
    if ( FAILED(hRet) ) return hRet;

    // Copy the filename over
    _tcscpy( m_strActorName, FileName );

    // Success!!
    return D3D_OK;
}
```

We pass the D3DX loader function the name of the file, the mesh creation options, the device, and a pointer to our ID3DXAllocateHierarchy derived object. The four member function of this object will be used as callback functions by D3DXLoadMeshHierarcdhyFromX. NULL is passed in as the fifth parameter, as we have no need to parse custom data objects. If we did, this is where we would pass in a pointer to our ID3DXLoadUserData derived object. For the sixth parameter, we pass in the address of the actor root frame pointer. On function return, this will point to the root frame of the hierarchy. Finally, we pass in NULL as the last parameter as we have not yet covered the ID3DXAnimationController interface. This will be covered in the next chapter.

If the D3DXLoadMeshHierarchyFromX function is successful, this function copies the filename passed in and stores it in the CActor::m_strActorName member variable. This will be used for actor lookups by the scene to avoid loading multiple actors with the same X file data.

CActor::SetWorldMatrix

Once the CActor has had its data loaded via the above function, it is ready to be positioned in the world. We can set the world space orientation and position of the actor (and all its child frames and meshes) using the SetWorldMatrix function.

This function should be passed a matrix describing the new world space transformation matrix for the actor. We can think of this as actually positioning the root frame in the world. Essentially, the passed matrix will become the root frame's parent frame of reference.

Our CScene class will store each CActor pointer in a CObject along with a world matrix. Therefore, when the scene updates the position of the CObject, its world matrix is passed into its CActor's SetWorldMatrix function to generate all the absolute world matrices for each frame. If no valid matrix is passed, then we will set the actor's matrix to identity, for safety.

```
void CActor::SetWorldMatrix( const D3DXMATRIX * mtxWorld, bool UpdateFrames  )
{
    // Store the currently set world matrix
    if ( mtxWorld )
        m_mtxWorld = *mtxWorld;
    else
        D3DXMatrixIdentity( &m_mtxWorld );

    // Update the frame matrices
    if ( UpdateFrames ) UpdateFrameMatrices( m_pFrameRoot, mtxWorld );
}
```

The function also takes a Boolean parameter called UpdateFrames which tells it whether you would like the traversal of the hierarchy and the updating of frame matrices to be performed immediately. If you pass 'true' then the CActor::UpdateFrameMatrices method will be called to traverse the hierarchy and update all the absolute world matrices of each frame. After the call to UpdateFrameMatrices, the absolute matrix in each frame contains the world matrix that should be set on the device in order to render any attached meshes. If 'false' is passed, the input matrix is simply stored in the actor's member variable and the update process can be deferred until a later time.

CActor::UpdateFrameMatrices

This function is responsible for stepping through the hierarchy in a recursive fashion and generating the absolute world matrices for each frame (stored in D3DXFRAME_MATRIX::mtxCombined). It can be called by the application, but is usually called by the SetWorldMatrix function above to recalculate the frame matrices when a new world matrix has been set.

The function is usually passed a pointer to the root frame as its first parameter. Its second parameter is the world space transformation matrix to be applied. The function recursively calls itself, so the changes filter down the hierarchy.

If you examine the SetWorldMarix call above, you can see that it calls this function passing in the root frame and the new world space transformation for the root frame (essentially the position of the actor in the scene). Let us now see what the function does with that information.

```
void CActor::UpdateFrameMatrices( LPD3DXFRAME pFrame, const D3DXMATRIX * pParentMatrix )
{
    D3DXFRAME_MATRIX * pMtxFrame = (D3DXFRAME_MATRIX*)pFrame;

    if ( pParentMatrix != NULL)
        D3DXMatrixMultiply( &pMtxFrame->mtxCombined,
                            &pMtxFrame->TransformationMatrix,
                            pParentMatrix);
    else
        pMtxFrame->mtxCombined = pMtxFrame->TransformationMatrix;

    // Move onto sibling frame
    if ( pMtxFrame->pFrameSibling )
         UpdateFrameMatrices( pMtxFrame->pFrameSibling, pParentMatrix );

    // Move onto first child frame
    if ( pMtxFrame->pFrameFirstChild )
         UpdateFrameMatrices( pMtxFrame->pFrameFirstChild, &pMtxFrame->mtxCombined );
}
```

First, the function casts the passed D3DXFRAME to its correct derived type. This is D3DXFRAME_MATRIX, which you will recall contains an extra matrix member (mtx_Combined) to store the world matrix for each frame during update traversals.

The function recursively calls itself until every frame in the hierarchy has been visited and its absolute world matrix (stored in mtx_Combined) has been calculated. After the function has returned program flow back to the caller, the combined matrix stored in each frame structure will contain the world matrix of that frame that should be set on the device in order to render any meshes attached to that frame.

The passed matrix is combined with the relative matrix stored in that frame to calculate its world matrix. If the frame has a sibling, then the function is called for its sibling too, so that it has a chance to combine its relative matrix with the passed matrix also. As the siblings are stored as a linked list, this causes all siblings of the frame to be visited and have their world matrices generated.

The next and final line of the function is very important to understanding the matrix concatenation process. If the frame has a child frame, then the function is called again for the child frame. Only this time, the matrix passed in is the world matrix that has just been generated for the current frame (i.e., mtx_Combined). It is the world matrix of the parent frame which describes the child frame's frame of reference. In the instance of the function that processes the child, its relative matrix is combined with the parent's **world** matrix, generating the world matrix for the child frame also. As the child position and orientation is described by its relative matrix in the parent's frame of reference, this will correctly position the child frame in world space such that it is correctly offset from its parent. If the parent matrix had more than one child frame, then the children will be connected in a linked list by their sibling pointers. In this case, the instance of the function processing the child will also cause each sibling to be processed using the same parent world matrix.

This function is actually very small but does a lot of work due to its recursive nature. Study this code and make sure you understand it.

CActor::DrawActorSubset

Because the CActors in our application have their AttributeID callback function registered, all CTriMeshes in the hierarchy are created in non-managed mode. This means the CTriMeshes in the scene have their attribute buffers filled with scene global attribute IDs. As the CTriMeshes themselves do not contain an attribute table in this mode, it is the responsibility of the scene object to render any CActors. This is done in the CScene::Render method which we will look at in a moment.

The scene must loop through all attributes it has in its attribute array and call the CActor::DrawSubset method for each actor. The scene must make sure that before it does, it sets the correct texture and material that is mapped to that attribute ID. This is virtually no different from how the scene rendered standalone CTriMeshes in non-managed mode in the previous lesson. The only difference is that the scene calls CActor::DrawActorSubset instead of CTriMesh::DrawSubset.

The DrawActorSubset method filters the rendering request down to each of its meshes. It does so by calling the CActor::DrawFrame method, passing in the subset number for which rendering has been requested. The DrawFrame method is another recursive function that visits every frame in the hierarchy searching for mesh containers. When a mesh container is found attached to a frame, the frame's world matrix (mtx_Combined) is set on the device, CTriMesh::DrawSubset is called for each mesh container attached to that frame, and traversal continues until every mesh has had a chance to render the requested subset.

As all of this traversal work is recursively performed inside the DrawFrame method, the DrawActorSubset method is a simple wrapper around the DrawFrame call. We will cover the DrawFrame method in a moment.

```
void CActor::DrawActorSubset( ULONG AttributeID )
{
    // Draw the frame heirarchy
    DrawFrame( m_pFrameRoot, (long)AttributeID );
}
```

CActor::DrawActor

If the CActor has not has the CALLBACK_ATTRIBUTEID function registered, but instead has had either CALLBACK_TEXTURE or CALLBACK_EFFECT registered, all meshes in the hierarchy will be created in managed mode. When this is the case, we can essentially describe the actor as being in managed mode.

In managed mode, each CTriMesh in the hierarchy will contain an attribute table containing the texture and material that should be set for each subset. This means each mesh in the hierarchy has the ability to render itself in a self-contained manner and does not require the scene to set the textures and materials

on the device prior to subset rendering. Just as CTriMesh has a function called CTriMesh::Draw which can be used render a managed mode CTriMesh in its entirety, so too does the actor.

The CActor::DrawActor function should only be used when the actor is in managed mode. Like the above function, it actually just wraps a call to the DrawFrame function. But unlike the DrawActorSubset call, no subset ID is passed into it. When the DrawFrame function has not been passed an attribute ID, it will render any CTriMeshes that exist in the hierarchy using the CTriMesh::Draw function. We know that this will cause the mesh to render all its subsets, taking care of state changing between each one by itself. Obviously, this can cause many redundant state changes, but managed mode is easy to use if you simply wish to view an asset that you have created or have not yet written the attribute callback function.

```
void CActor::DrawActor( )
{
    // Draw the frame heirarchy
    DrawFrame( m_pFrameRoot );
}
```

CActor::DrawFrame

It is in this function (called by both functions discussed previously) that the main rendering traversal is done. It is recursively called such that it processes every frame in the hierarchy.

For the current frame being processed (which is initially the root when called by the prior functions) its mesh container pointer will be non-NULL if a mesh container is attached. If multiple mesh containers have been assigned to this frame, then each mesh container will be arranged in a linked list via the D3DXMESHCONTAINER::pNextMeshContainer pointer.

If the mesh container pointer is not NULL, then a mesh exists and will need to be rendered. We set the frame world matrix, mtx_Combined (generated by a prior call to SetWorldMatrix or UpdateFrameMatrices), as the world matrix on the device. This matrix describes the world transform for all meshes in this frame.

We next initialize a for loop to step through all mesh containers assigned to this frame. For each one found, we call the CActor::DrawMeshContainer method to render the CTriMesh. How this function operates depends on the attribute ID passed into the function. If an attribute ID has been passed, then DrawMeshContainer will call CTriMesh::DrawSubset using this ID to render the requested subset stored in the mesh. If no attribute ID has been passed, as is the case when this function is called by the DrawActor method, the actor is assumed to be in managed mode and the DrawMeshContainer will render its mesh using CTriMesh::Draw to render all subsets at once.

```
void CActor::DrawFrame( LPD3DXFRAME pFrame, long AttributeID /* = -1 */ )
{
    LPD3DXMESHCONTAINER pMeshContainer;
    D3DXFRAME_MATRIX  * pMtxFrame  = (D3DXFRAME_MATRIX*)pFrame;

    // Set the frames combined matrix
    if (pMeshContainer) m_pD3DDevice->SetTransform( D3DTS_WORLD, &pMtxFrame->mtxCombined );
```

```
    // Loop through the frame's mesh container linked list
    for ( pMeshContainer = pFrame->pMeshContainer;
          pMeshContainer;
          pMeshContainer = pMeshContainer->pNextMeshContainer )
    {
        // Draw this container
        DrawMeshContainer( pMeshContainer, pFrame, AttributeID );

    } // Next Container

    // Move onto next sibling frame
    if ( pFrame->pFrameSibling ) DrawFrame( pFrame->pFrameSibling, AttributeID );

    // Move onto first child frame
    if ( pFrame->pFrameFirstChild ) DrawFrame( pFrame->pFrameFirstChild, AttributeID );
}
```

The familiar lines at the bottom of this function allow us to step through any sibling frames attached to the current frame, and the step down a level in the hierarchy to the child frames, if they exist. At the end of this function, all frames will have been visited. If an attribute ID was passed (non-managed mode), then all mesh subsets in the hierarchy with the requested ID will have been rendered. If no attribute ID is passed (-1 is the default) then the actor is in managed mode and all meshes in the hierarchy will have been entirely rendered.

CActor::DrawMeshContainer

The DrawMeshContainer function is called once by the previous function for each mesh container that exists in the tree. A mesh container linked to a frame contains a single CTriMesh, but there may be multiple mesh containers linked to a single frame. The job of this function, as we saw above, is to render the CTriMesh stored within each mesh container appropriately.

```
void CActor::DrawMeshContainer( LPD3DXMESHCONTAINER pMeshContainer,
                                LPD3DXFRAME pFrame, long AttributeID /* = -1 */ )
{
    D3DXFRAME_MATRIX          * pMtxFrame  = (D3DXFRAME_MATRIX*)pFrame;
    D3DXMESHCONTAINER_DERIVED * pContainer = (D3DXMESHCONTAINER_DERIVED*)pMeshContainer;
    CTriMesh                  * pMesh      = pContainer->pMesh;

    // Validate requirements
    if ( !pMesh ) return;

    // Render the mesh
    if ( AttributeID >= 0 )
    {
        // Set the FVF for the mesh
        m_pD3DDevice->SetFVF( pMesh->GetFVF() );
        pMesh->DrawSubset( (ULONG)AttributeID );

    } // End if attribute specified
    else
    {
        pMesh->Draw( );

    } // End if no attribute specified
}
```

This function is very simple thanks to the fact that we are using our CTriMesh class to represent mesh data. First it fetches a pointer to the CTriMesh object stored in the mesh container. Next, a check is done on the AttributeID passed in. If it is less than zero (-1) then this actor is in managed mode and the mesh can render itself (setting its own texture and material states before rendering each of its subsets). When this is the case, the function simply calls CTriMesh::Draw to instruct the mesh to render itself entirely.

If this is not the case, and the passed attribute ID is larger than –1, then the scene is rendering the individual mesh subsets, and all we wish to render is the requested subset of the mesh stored here. So we set the correct FVF flags for the mesh and use the CTriMesh::DrawSubset method to render only the requested subset. Notice that in this implementation of the function, the passed frame pointer is not used. We may very well need this later on though, and it is very useful for the rendering function to have access to its parent.

CActor Accessor Functions

The CActor class exposes several state accessor function so the application can performs tasks such as: retrieve a pointer to one of its registered callback functions, inquire about its mesh creation functions, or retrieve the name of the actor (i.e., the name of the X file).

```
CALLBACK_FUNC CActor::GetCallback( CALLBACK_TYPE Type ) const
{
    return m_CallBack[Type];
}
```

```
ULONG CActor::GetOptions( ) const
{
    return m_nOptions;
}
```

```
LPCTSTR CActor::GetActorName( ) const
{
    return m_strActorName;
}
```

CActor::GetFrameByName

This method is used to search an actor’s hierarchy for a frame with a matching name. The function will return a pointer to this frame if a name match is found. You will recall that in the X file, frames will often be assigned a name, and this is vital if we intend to animate them. We will see in our demo how the scene uses this function, after the automobile has been loaded, to retrieve pointers to the wheel and pivot frames. We then attach the relative matrices of these frames to CAnimation objects that can modify the matrices and allow us to spin the wheels.

The function is passed a string containing the name of the frame we wish to find, followed by a pointer to a frame from which you wish the search to begin. This will usually be the root frame so that the entire hierarchy is searched, but alternatively, we could limit our search to only a given sub-tree if desired.

```
LPD3DXFRAME CActor::GetFrameByName( LPCTSTR strName, LPD3DXFRAME pFrame /* = NULL */ ) const
{
    LPD3DXFRAME pRetFrame = NULL;

    // Any start frame passed in?
    if ( !pFrame ) pFrame = m_pFrameRoot;

    // Does this match ?
    if ( _tcsicmp( strName, pFrame->Name ) == 0 ) return pFrame;

    // Check sibling
    if ( pFrame->pFrameSibling )
    {
        pRetFrame = GetFrameByName( strName, pFrame->pFrameSibling );
        if ( pRetFrame ) return pRetFrame;

    } // End if has sibling

    // Check child
    if ( pFrame->pFrameFirstChild )
    {
        pRetFrame = GetFrameByName( strName, pFrame->pFrameFirstChild );
        if ( pRetFrame ) return pRetFrame;

    } // End if has sibling

    // Nothing found
    return NULL;
}
```

It should be obvious by now that this will be another recursive function because of its need to traverse the hierarchy. Starting at the frame passed in, we compare its name against the passed name. If the frame name matches the string name then we have found our match and return this frame immediately.

If it does not match then a check of the sibling pointer of the current frame is made and the function is recursively called if a sibling exists. We know this will, in turn, cause all siblings in the linked list to be visited and their names checked. When program flow returns back to the current instance, if pRetFrame is not NULL then it means that one of the sibling frames has a name that matched, so we return it. Otherwise, we recursively check the children in the same way.

If we get to the end of the function and there is no frame in the hierarchy with a matching name we return NULL. We will see this function being used to connect the CAnimation objects to the various frames in the hierarchy when we examine the modifications to the CScene class.

CActor::SaveActorToX

It is sometimes useful to be able to save the contents of an actor out to an X file. Perhaps you have altered the data in some way and would like the changes to be preserved. While it may not be an extremely useful option within your actual game, having this ability will certainly come in useful when you are writing your own tools (e.g., a hierarchy editor). In fact, in our next workbook we will build just such a tool (called Animation Splitter) that uses the CActor object to load an X file, split any animation data stored in that X file into multiple animations, and then save the re-organized actor and animation data back out to the X file as a hierarchy. You will understand why we build such a tool in the next

chapter when we discuss the D3DX animation system. For the time being, just know that there may be times when you will want your actor's frame hierarchy saved to an X file.

Before we look at the CActor::SaveActorToX method's source code which is a mere few lines in length, let us first take a look at the global D3DX helper function it uses to perform the task.

The D3DXSaveMeshHierarchyToFile Function

The D3DX library exposes a global function that an application can use to save an entire frame/mesh hierarchy out to an X file. It can be thought of as the opposing function to the D3DXLoadMeshHierarchyFromX function in that information flows from the hierarchy in memory out to a new X file, whereas in the case of the D3DXLoadMeshHierarchyFromX function, the information flow is in the other direction. The method is shown below along with a description of each of its parameters.

```
HRESULT WINAPI D3DXSaveMeshHierarchyToFile
(
    LPCSTR pFilename,
    DWORD XFormat,
    const D3DXFRAME *pFrameRoot,
    LPD3DXANIMATIONCONTROLLER pAnimController,
    LPD3DXSAVEUSERDATA pUserDataSaver
);
```

LPCSTR pFilename
This is a string containing the name of the new X file that will be created.

DWORD XFormat
This parameter is used to tell the function whether the application would like the X file to be created as a text or binary X file. We can pass one of the following D3DXF_FILEFORMAT flags:

#define	Value	Description
D3DXF_FILEFORMAT_BINARY	0	Legacy-format binary file.
D3DXF_FILEFORMAT_COMPRESSED	2	Compressed file. May be used in combination with other D3DXF_FILEFORMAT_ flags. This flag is not sufficient to completely specify the saved file format.
D3DXF_FILEFORMAT_TEXT	1	Text file.

const D3DXFRAME *pFrameRoot
All that is needed to gain access to the frame hierarchy for saving is the root frame. This function can use the root to traverse down through the tree and save off any frames, matrices, meshes, texture filenames and materials it finds. The result will be an X file containing all the information used by the hierarchy.

LPD3DXANIMATIONCONTROLLER *pAnimController*
We will see in the next chapter that when an X file that contains animation data is loaded, the D3DXLoadMeshHierachyFromX function will return a pointer an ID3DXAnimationController interface. We can think of the animation controller as being a manager object that contains all the animation data used by the hierarchy. It presents the application with a means to play back those animations via its exposed methods. For now, our actor will pass in NULL for this parameter as it does not yet support animation data. In the next workbook, we will upgrade our actor so that it also includes animation support. When this is the case, the save function must also be able to save off any animation data to the X file. Therefore, in the next chapter, when we upgrade our actor, this parameter will be passed a pointer to the actor's animation controller interface.

LPD3DXSAVEUSERDATA *pUserDataSaver*
When loading an X file using D3DXLoadMeshHierarchyFromX, we have the option of passing in a pointer to an ID3DXLOADUSERDATA derived object, whose methods would be called by D3DX when custom data objects are discovered in the X file. When saving the X file, D3DX will require the application to write out custom data chunks as well. Thus, the abstract ID3DXSAVEUSERDATA base class is the complementary interface for saving. Although we will not use this parameter in our demo, if you open up d3dxanim.h you will see that it is an interface with two methods that must be implemented in the derived class. We will simply pass in NULL as we have no custom chunks in our current application.

Finally, we will look at the code to the CActor::SaveActorToX method:

```
HRESULT CActor::SaveActorToX( LPCTSTR FileName, ULONG Format )
{
    HRESULT hRet;

    // Validate parameters
    if ( !FileName ) return D3DERR_INVALIDCALL;

    // If we are NOT managing our own attributes, fail
    if ( GetCallback( CActor::CALLBACK_ATTRIBUTEID ).pFunction != NULL )
        return D3DERR_INVALIDCALL;

    // Save the hierarchy back out to file
    hRet = D3DXSaveMeshHierarchyToFile( FileName, Format, m_pFrameRoot, NULL, NULL );

    if ( FAILED(hRet) ) return hRet;

    // Success!!
    return D3D_OK;
}
```

The application passes in the name of the X file and the format (binary vs. text) of the file. This request is passed straight into the D3DXSaveMeshHierarchyToFile function, along with the root frame of the actor. Notice that the function returns failure if the actor is in non-managed mode. The reason for this will be discussed in more detail later in this workbook when we cover the CAllocateHierarchy::CreateMeshContainer method. Ultimately, in non-managed mode, the application may have remapped all the subset IDs into a global pool of textures and materials. Because of this, the D3DXSaveMeshHierarchyToFile method will no longer have access to the materials and texture filenames used by each subset in a mesh. Therefore, saving a non-managed actor would result in meshes

being written out with incorrect texture and material data. As you are only likely to use the CActor::SaveActorToX in a situation where it is being used as part of a tool you are developing, it is a limitation we can work with. Unlike a real-time game situation, using managed mode actors in a tool is acceptable since performance is not necessarily the top priority.

Source Code Walkthrough – CAllocateHierarchy

The CAllocateHierarchy class is derived from the abstract interface ID3DXAllocateHierarchy. This interface exposes four virtual functions which must be overridden in the derived class. These four functions are used as callbacks during the creation and destruction of the frame hierarchy.

The CActor temporarily instantiates an object of this type in the CActor::LoadActorFromX function and passes it to D3DXLoadMeshHierarchyFromX. This function will use the object's CreateFrame and CreateMeshContainer functions to allow the application to allocate the frames and mesh containers that will be attached to the hierarchy. The CActor instantiates an object of this type a second time in the CActor::Release function and passes it to the D3DXFrameDestroy function to unload the hierarchy from memory. This function uses the interface's DestroyFrame and DestroyMeshContainer methods to allow the application to release the frames and meshes in the hierarchy and perform any other clean up.

The derived class definition (see CActor.h) is shown below. It implements the four base class virtual functions and exposes a constructor that accepts a pointer to a CActor. The CActor pointer is stored in its only member variable so that the CreateMeshContainer method can access the registered callbacks of the actor during material processing.

```
class CAllocateHierarchy: public ID3DXAllocateHierarchy
{
public:
    // Constructors & Destructors for This Class
    CAllocateHierarchy( CActor * pActor ) : m_pActor(pActor) {}


    // Public Functions for This Class
    STDMETHOD(CreateFrame)          ( THIS_ LPCTSTR Name, LPD3DXFRAME *ppNewFrame);
    STDMETHOD(CreateMeshContainer)  ( THIS_ LPCTSTR Name, CONST D3DXMESHDATA *pMeshData,
                                      CONST D3DXMATERIAL *pMaterials,
                                      CONST D3DXEFFECTINSTANCE *pEffectInstances,
                                      DWORD NumMaterials, CONST DWORD *pAdjacency,
                                     LPD3DXSKININFO pSkinInfo,
                                     LPD3DXMESHCONTAINER *ppNewMeshContainer);

    STDMETHOD(DestroyFrame)         (THIS_ LPD3DXFRAME pFrameToFree);
    STDMETHOD(DestroyMeshContainer) (THIS_ LPD3DXMESHCONTAINER pMeshContainerBase);


    // Public Variables for This Class
    CActor  * m_pActor;         // Actor we are allocating for

};
```

CAllocateHierarchy::CreateFrame

The CreateFrame function is called by D3DXLoadMeshHierarchyFromX when a new frame data object is encountered in the X file. The name of the frame will be passed to this function as its first parameter along with the address of a D3DXFRAME pointer.

CreateFrame must allocate the frame structure and perform any initialization of its data members before assigning the passed frame pointer to the newly allocated frame structure. When the function returns, D3DX will attach this frame to the hierarchy in its correct position.

It is very useful that D3DX provides us with a means for creating the frame structure and does not allocate a D3DXFRAME structure on our behalf. This allows us to allocate our D3DXFRAME derived structure type, which contains an extra matrix (mtx_Combined) to store the absolute world matrix of the frame when the hierarchy is updated (via CActor::UpdateFrameMatrices). Because our D3DXFRAME_MATRIX structure is derived from D3DXFRAME, the variables that D3DX cares about (basically just the matrix, sibling, and child pointers) are still in the correct position. So the structure can be safely cast. From the perspective of D3DX, it is passed back a D3DXFRAME structure and it can treat it as such.

```
HRESULT CAllocateHierarchy::CreateFrame( LPCTSTR Name, LPD3DXFRAME *ppNewFrame )
{
    D3DXFRAME_MATRIX * pNewFrame = NULL;

    // Clear out the passed frame (it may not be NULL)
    *ppNewFrame = NULL;

    // Allocate a new frame
    pNewFrame = new D3DXFRAME_MATRIX;
    if ( !pNewFrame ) return E_OUTOFMEMORY;

    // Clear out the frame
    ZeroMemory( pNewFrame, sizeof(D3DXFRAME_MATRIX) );

    // Copy over, and default other values
    if ( Name ) pNewFrame->Name = _tcsdup( Name );
    D3DXMatrixIdentity( &pNewFrame->TransformationMatrix );

    // Pass this new pointer back out
    *ppNewFrame = (D3DXFRAME*)pNewFrame;

    // Success!!
    return D3D_OK;
}
```

The function starts by allocating our derived D3DXFRAME_MATRIX structure, initializing it to zero for safety. It also sets the frame's relative matrix to identity. This is also for safety, since this matrix should be overwritten by D3DX with the actual matrix values for this frame stored in the X file when the function returns.

The next thing we do is store the name of the frame (passed as the first parameter) in the frame structure so that we can traverse the actor's hierarchy and search for frames by name. This is very important

because in our demo application we will need to search the actor's hierarchy for the wheel and pivot frames so that we can attach CAnimation objects to them.

Notice that we cannot simply copy the passed string into the frame structure Name member. This string memory is owned by D3DX and will be released as soon as the function returns. This would leave us with an invalid string pointer in our frame. Therefore we must copy the contents of the string into a new string that we allocate. The _tcsdup function does this job for us. We just pass in the string we wish to copy and it will allocate a new string of the correct size and copy over the contents of the source string. We can then store the returned string in our frame.

Finally, we assign the D3DXFRAME pointer passed by D3DX to point to our newly allocated D3DXFRAME_MATRIX (making sure we cast it). On function return, D3DX will now have access to our frame and can use its sibling and child pointers to attach it to the hierarchy.

CAllocateHierarchy::CreateMeshContainer

The CreateMeshContainer function is called by D3DXLoadMeshHierarchyFromX whenever a mesh data object is encountered in the X file. This method must use the passed information to construct a D3DXMESHCONTAINER (or derived) structure and return it to D3DX.

We are using our own derived mesh container type (called D3DXMESHCONTAINER_DERIVED) which includes an additional CTriMesh pointer. With D3DX providing this mesh creation callback mechanism, we are afforded complete freedom to manage the mesh data and materials for the mesh in whatever way our application sees fit. This will allow us to store the passed mesh data in a CTriMesh and send the material data to the scene callbacks for storage. This function also provides us with a chance to check the creation flags of the mesh and its vertex format and clone it into a new mesh using the format we desire.

Because this function actually has quite a lot of work to do, we will cover it a section at a time. Luckily, the resource management section is almost an exact duplicate of the material handling code we saw in the previous workbook for CTriMesh. This code should be very familiar to you.

```
HRESULT CAllocateHierarchy::CreateMeshContainer(LPCTSTR Name,
                                                CONST D3DXMESHDATA*pMeshData,
                                                CONST D3DXMATERIAL *pMaterials,
                                                CONST D3DXEFFECTINSTANCE *pEffectInstances,
                                                DWORD NumMaterials,
                                                CONST DWORD *pAdjacency,
                                                LPD3DXSKININFO pSkinInfo,
                                                LPD3DXMESHCONTAINER *ppNewMeshContainer)
{

    ULONG                       AttribID, i;
    HRESULT                     hRet;
    LPD3DXMESH                  pMesh          = NULL;
    D3DXMESHCONTAINER_DERIVED *pMeshContainer = NULL;
    LPDIRECT3DDEVICE9           pDevice        = NULL;
    CTriMesh                   *pNewMesh       = NULL;
    MESH_ATTRIB_DATA           *pAttribData    = NULL;
    ULONG                      *pAttribRemap   = NULL;
```

```
    bool                        ManageAttribs  = false;
    bool                        RemapAttribs   = false;
    CALLBACK_FUNC               Callback;

    // We only support standard meshes (i.e. no patch or progressive meshes in this demo)
    if ( pMeshData->Type != D3DXMESHTYPE_MESH ) return E_FAIL;
```

We can see immediately that the function accepts quite a lengthy parameter list. As the first parameter we are passed the name of the mesh. This is the name assigned to the mesh data object in the X file, if one exists. As the second parameter we are passed a D3DXMESHDATA structure. You will recall from the textbook that this structure contains either an ID3DXMesh interface pointer, an ID3DXPMesh interface pointer, or an ID3DXPatchMesh interface pointer. It also contains a Type member variable describing which mesh type it is. The D3DXMESHDATA member is shown below:

```
typedef struct D3DXMESHDATA
{
    D3DXMESHDATATYPE Type;
    union
    {
      LPD3DXMESH           pMesh;
      LPD3DXPMESH         pPMesh;
      LPD3DXPATCHMESH pPatchMesh;
    }
} D3DXMESHDATA, *LPD3DXMESHDATA
```

D3DXMESHDATATYPE is an enumeration which can contain one of three values: D3DXMESHTYPE_MESH, D3DXMESHTYPE_PMESH or D3DXMESHTYPE_PATCHMESH.

In our demo code, we will only support the common case where the structure contains an ID3DXMesh pointer. If you wish to add support for loading progressive meshes from an X file, then you should be able to modify this function to handle both cases without any problem given our discussions in the last chapter.

Returning to the above code, we can see that the third parameter to this function is an array of D3DXMATERIAL structures containing the texture filename and material information for each subset in the passed mesh. The fourth parameter contains an array of effect instances. We will ignore any reference to this at the moment until we cover effect files in Module III. The fifth parameter describes the number of elements in the previous two arrays (the number of subsets in the mesh). This is followed by a pointer to the mesh adjacency information. The penultimate parameter is a pointer to a D3DXSKININFO structure. We will see this being used in Chapter 11 when we cover skinned meshes and skeletal animation. For now we can ignore this member. The final parameter is the address of a pointer to a D3DXMESHCONTAINER structure which our function will assign to the mesh container structure that we allocate and populate within this function. When the function returns, D3DX will use this pointer to connect the mesh container to its correct frame in the hierarchy.

In the next code section we see that the first thing we do is assign the ID3DXMesh interface stored in the passed D3DXMESHDATA member to a local member variable (pMesh) for easier access. Notice that although we are copying a pointer to an interface here, we are not incrementing the reference count. This is because we may not actually want to keep this mesh. We have to first check its creation parameters and its vertex format.

Currently we only support FVF style mesh creation, so if this mesh's FVF flags are zero, then we return from the function.

```
    // Extract the standard mesh from the structure
    pMesh = pMeshData->pMesh;

    // We require FVF compatible meshes only
    if ( pMesh->GetFVF() == 0 ) return E_FAIL;

    // Allocate a mesh container structure
    pMeshContainer = new D3DXMESHCONTAINER_DERIVED;
    if ( !pMeshContainer ) return E_OUTOFMEMORY;

    // Clear out the structure to begin with
    ZeroMemory( pMeshContainer, sizeof(D3DXMESHCONTAINER_DERIVED) );

    // Copy over the name
    if ( Name ) pMeshContainer->Name = _tcsdup( Name );
```

You can see above that we allocate our new mesh container. It will be a D3DXMESHCONTAINER_DERIVED structure which has a CTriMesh pointer that we will need to use. We initialize the structure to zero and then copy over the name of the mesh into the mesh container's Name member. As described in the CAllocateHierarchy::CreateFrame method, the memory containing the string is owned by D3DX, so we must make a copy because we cannot rely on its persistence after the function returns.

At this point we have an empty mesh container, so it is time to start filling it up with the passed data. Before we do that however, we know that we want our data to be stored in a CTriMesh object, so we will allocate one. To avoid confusion, the point should be made that the local variable pNewMesh is a pointer to a CTriMesh object. The local variable pMesh is a pointer to an ID3DXMesh interface.

```
    // Allocate a new CTriMesh
    pNewMesh = new CTriMesh;
    if ( !pNewMesh ) { hRet = E_OUTOFMEMORY; goto ErrorOut; }
```

We now have an empty CTriMesh in hand. In the previous workbook, it is at this point that we would register our callback functions, but that would be pointless in this code because the CTriMesh::LoadMeshFromX function will never be used. That is why the same callback code will need to be implemented by hand in this function.

The ID3DXMesh data has already been loaded and passed to us by D3DX along with the materials that it uses. We will see in a moment that we will attach this ID3DXMesh to the CTriMesh using the CTriMesh::Attach function. But before we do that, we want to make sure that the ID3DXMesh that we have been passed is using a data format our application is happy with. If not, we will need to clone it into the appropriate format.

The next section of code tests to see whether the FVF flags of the ID3DXMesh that D3DX has passed us contains vertex normals. If it does not, then this mesh is no good to us and we will clone a new mesh and add the normals ourselves. We also clone the mesh if the mesh creation options are not what we requested. This may seem a strange test to do; after all, the application will pass the desired mesh

creation options into the CActor::LoadActorFromX function, which in turn passes the options on to the D3DXLoadMeshHierarchyFromX function. Therefore, the meshes should surely be created using the options we initially registered with the actor. Well, you would certainly think so. However, in our laboratory tests on various machines we found that the meshes created by D3DX did not always match the mesh creation options passed into D3DXLoadMeshHierarchyFromX. Sometimes it even placed the vertex or index buffer in system memory. So to be safe, we will retrieve the options from the passed mesh and compare them against the options registered with the actor (by the application), and if they are not a match, we will clone the mesh using the correct mesh creation flags.

```
    // If there are no normals, this demo requires them, so add them to the mesh's FVF
    if (!(pMesh->GetFVF()&D3DFVF_NORMAL) || (pMesh->GetOptions()!=m_pActor->GetOptions()) )
    {
        LPD3DXMESH pCloneMesh = NULL;

        // Retrieve the mesh's device (this adds a reference)
        pMesh->GetDevice( &pDevice );

        // Clone the mesh
        hRet = pMesh->CloneMeshFVF( m_pActor->GetOptions(),
                                    pMesh->GetFVF() | D3DFVF_NORMAL,
                                    pDevice,
                                    &pCloneMesh );

        if ( FAILED( hRet ) ) goto ErrorOut;

        // Note: we don't release the old mesh here, because we don't own it
        pMesh = pCloneMesh;

        // Compute the normals for the new mesh if there was no normal to begin with
        if ( !(pMesh->GetFVF() & D3DFVF_NORMAL) ) D3DXComputeNormals( pMesh, pAdjacency );

        // Release the device, we're done with it
        pDevice->Release();
        pDevice = NULL;

        // Attach our specified mesh to the new mesh (this addref's the chosen mesh)
        pNewMesh->Attach( pMesh );


    } // End if no vertex normal or wrong options
    else
    {
        // Simply attach our specified mesh to the new mesh (this addref's the chosen mesh)
        pNewMesh->Attach( pMesh );

    } // End if options ok
```

In the above code you can see that if the mesh creation options are incorrect or normals do not exist in the vertices then we clone a new mesh using the correct format. We never release the original mesh interface because we never incremented its reference count earlier. At this point, the original mesh (which we no longer need) has a reference count of 1 (i.e., D3DX's claim on the interface). When the function returns and D3DX decrements the reference count of the original mesh, it will be unloaded from memory. Notice that after the clone operation, we reassign the local pMesh pointer to point at the cloned mesh interface instead.

If normals did not previously exist then we just added room for them. Since the normal data is not yet initialized, we use the D3DXComputeNormals function, passing in the adjacency information that we were passed by D3DX, to calculate the normals.

Next, we release the device interface (retrieved from the mesh) which we no longer need, and we call the CTriMesh::Attach method to attach the tri-mesh to our ID3DXMesh. We looked at the CTriMesh::Attach function in the previous workbook -- it assigns the CTriMesh internal ID3DXMesh pointer to point to the passed interface. The final section of the above code demonstrates that when the passed mesh is already in the correct format, we can simply attach it to the CTriMesh. The Attach method will automatically increment the reference count of the passed interface.

At this point we have a CTriMesh with a valid ID3DXMesh containing the mesh data from the X file. We now have to decide what to do with the material data passed into the function. If the actor is in managed mode, then we will wish to register the texture filenames for each subset with the texture callback (pre-registered with the actor) and store the texture pointers and materials in the mesh itself. If this is a non-managed mode actor, then we will need to perform attribute buffer remapping.

The first line of the following code tests to see if the actor has had its CALLBACK_ATTRIBUTEID function registered. If so, then this actor is in non-managed mode and the meshes will contain their own attribute lists. If the function pointer is NULL, then ManageAttribs will be set to true and the actor is in managed mode. When this is the case, our first task is to allocate the CTriMesh's attribute data array to the correct size -- we need to it to be large enough to contain an entry for each subset. We allocate this array using the CTriMesh::AddAttributeData function and pass in the number of subsets in the mesh. Once it is allocated we use CTriMesh::GetAttributeData to retrieve a pointer to the first element in the array. This pointer will be used to populate the array with texture pointers and material information.

```
    // Are we managing our own attributes ?
    ManageAttribs = (m_pActor->GetCallback
                    ( CActor::CALLBACK_ATTRIBUTEID ).pFunction == NULL);

    // Allocate the attribute data if this is a manager mesh
    if ( ManageAttribs == true && NumMaterials > 0 )
    {
        if ( pNewMesh->AddAttributeData( NumMaterials ) < -1 )
                                                { hRet = E_OUTOFMEMORY; goto ErrorOut; }
        pAttribData = pNewMesh->GetAttributeData();

    } // End if managing attributes
    else
    {
        // Allocate attribute remap array
        pAttribRemap = new ULONG[ NumMaterials ];
        if ( !pAttribRemap ) { hRet = E_OUTOFMEMORY; goto ErrorOut; }

        // Default remap to their initial values.
        for ( i = 0; i < NumMaterials; ++i ) pAttribRemap[ i ] = i;

    } // End if not managing attributes
```

The above code also shows the case for a non-managed mode mesh. In this case we know we will need to re-map the attribute buffer, so we allocate an array large enough to be used as a temporary re-mapping buffer. This was all covered in the CTriMesh class in the previous workbook, so this should

not be new to you. We will use this temporary buffer to store the global attribute IDs for each subset returned by the scene's resource callback function.

We now loop through each material in the passed D3DXMATERIAL array and process them in accordance with the mode of the actor. The following code was described in its entirety in the previous chapter (only it was inside CTriMesh instead), so you should refer back if you need to refresh your memory. At the end of the loop, the attribute data array inside the CTriMesh will contain an element for each subset in the mesh. Each element will contain the texture, material and possible effect file used by the subset.

```
    // Loop through and process the attribute data
    for ( i = 0; i < NumMaterials; ++i )
    {
        if ( ManageAttribs == true )
        {
            // Store material
            pAttribData[i].Material = pMaterials[i].MatD3D;

            // Note : The X File specification contains no ambient material property
            pAttribData[i].Material.Ambient = D3DXCOLOR( 1.0f, 1.0f, 1.0f, 1.0f );

            // Request texture pointer via callback
            Callback = m_pActor->GetCallback( CActor::CALLBACK_TEXTURE );
            if ( Callback.pFunction )
            {
                COLLECTTEXTURE CollectTexture = (COLLECTTEXTURE)Callback.pFunction;
                pAttribData[i].Texture = CollectTexture( Callback.pContext,
                                                         pMaterials[i].pTextureFilename );

                // Add reference. We are now using this
                if ( pAttribData[i].Texture ) pAttribData[i].Texture->AddRef();

            } // End if callback available

            // Request effect pointer via callback
            Callback = m_pActor->GetCallback( CActor::CALLBACK_EFFECT );
            if ( Callback.pFunction )
            {
                COLLECTEFFECT CollectEffect = (COLLECTEFFECT)Callback.pFunction;
                pAttribData[i].Effect = CollectEffect( Callback.pContext,
                                                       pEffectInstances[i] );

                // Add reference. We are now using this
                if ( pAttribData[i].Effect ) pAttribData[i].Effect->AddRef();

            } // End if callback available

        } // End if attributes are managed
```

The next code block shows what happens when the actor is non-managed. Again, you can refer to the prior workbook for details about the operations encountered.

```
        else
        {
            // Request attribute ID via callback
            Callback = m_pActor->GetCallback( CActor::CALLBACK_ATTRIBUTEID );
            if ( Callback.pFunction )
```

```
            {
             COLLECTATTRIBUTEID CollectAttributeID =(COLLECTATTRIBUTEID)Callback.pFunction;
             AttribID = CollectAttributeID( Callback.pContext,
                                            pMaterials[i].pTextureFilename,
                                            &pMaterials[i].MatD3D, &pEffectInstances[i] );

                // Store this in our attribute remap table
                pAttribRemap[i] = AttribID;

                // Determine if any changes are required so far
                if ( AttribID != i ) RemapAttribs = true;

            } // End if callback available

        } // End if we don't manage attributes

    } // Next Material
```

At this point, a managed mesh will be completely set up, with its internal attribute information stored for self-contained rendering.

If the actor is in non-managed mode, then the current mesh still needs to have its attribute buffer re-mapped. So the next snippet of code will lock the attribute buffer of a non-managed mesh and use the temporary re-map array we just populated to change the attribute of each triangle to the new global attribute ID. This code is also not new to us -- the CTriMesh performs a very similar step when its CTriMesh::LoadMeshFromX function is used.

```
    // Remap attributes if required
    if ( pAttribRemap != NULL && RemapAttribs == true )
    {
        ULONG * pAttributes = NULL;

        // Lock the attribute buffer
        hRet = pMesh->LockAttributeBuffer( 0, &pAttributes );
        if ( FAILED(hRet) ) goto ErrorOut;

        // Loop through all faces
        for ( i = 0; i < pMesh->GetNumFaces(); ++i )
        {
            // Retrieve the current attribute ID for this face
            AttribID = pAttributes[i];

            // Replace it with the remap value
            pAttributes[i] = pAttribRemap[AttribID];

        } // Next Face

        // Finish up
        pMesh->UnlockAttributeBuffer( );

    } // End if remap attributes

    // Release remap data
    if ( pAttribRemap ) delete []pAttribRemap;
```

Our CTriMesh is now fully created, and the scene (or whatever object has registered the resource callbacks) has loaded and stored any textures and materials that the mesh uses.

We can now perform optimizations and cleaning using our CTriMesh methods. We will first perform a weld of the vertices to merge any duplicated vertices that may exist into a single vertex. This will allow us to reduce the vertex count in some cases. We then issue a call to CTriMesh::OptimizeInPlace to perform a compaction of the data (removing degenerate triangles for example), an attribute sort of the attribute buffer (for better subset rendering performance), and a vertex cache optimization to rearrange the mesh data for better vertex cache coherency.

```
    // Attempt to optimize the new mesh
    pNewMesh->WeldVertices( 0 );
    pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );
```

With our mesh in top condition, we can now assign it to the mesh container that we allocated earlier in the function. Recall that we actually allocated an instance of our derived mesh container class, which contains a CTriMesh pointer in addition to the usual members of the base class. We store the CTriMesh pointer to our new mesh in the mesh container and store the CTriMesh's underlying ID3DXMesh interface in the mesh container's D3DXMESHDATA member. We do this final step so that D3DX helper functions (e.g., D3DXComputeBoundingSphere) will still be able to work with the mesh data stored in our hierarchy. Remember that any D3DX helper function that deals with meshes in a hierarchy will not be aware that the hierarchy now contains mesh data in the proprietary CTriMesh type added to the derived class. These functions expect the mesh data to be stored as an ID3DXMesh interface inside the D3DXMESHDATA member of the mesh container.

By fetching the CTriMesh's underlying ID3DXMesh interface and storing it in the D3DXMESHDATA member of the mesh container, we keep everybody happy. Our actor can access the methods of CTriMesh using the mesh container's CTriMesh pointer, and D3DX can traverse the hierarchy and access the raw mesh data via the ID3DXMesh interface stored in the expected place.

One such D3DX helper function that will need to access the mesh data is the D3DXSaveMeshHierarchyToFile method which we discussed earlier (see CActor::SaveActorToX). This function will traverse the hierarchy and write the frame hierarchy information and mesh container information out to the X file. But this function needs more than just access to the mesh data; it also needs to know about the materials and texture filenames used by the mesh so that they can be written to the file as well. This would seem to present a bit of a problem since the application manages the loading an storing of textures in data areas the actor might not have access to. The solution is actually quite simple -- it is in this function that we are first passed the D3DXMATERIAL array that contains the texture filenames and materials for each subset. We have seen how the scene callback functions are invoked by the actor to turn information in this array into real textures (instead of just filenames). Therefore, at this point, we already know the materials and texture filenames used by the mesh because D3DX passed them into this function to help us load the correct resources. So when this X file is saved, we simply need D3DX to get access to this array of D3DXMATERIAL's that it originally passed us during the loading process.

As we saw earlier, the D3DXMESHCONTAINER structure has members to store both a pointer to a D3DXMATERIAL array and the number of materials used by this array. This is exactly where the D3DXSaveMeshHierarchyToFile function expects the material and texture data for each mesh container to be stored. So all we have to do to make the save function work, is make a copy of the D3DXMATERIAL array that was passed into this function and store that in the mesh container too. We

must make a copy of the material array, and not just assign the mesh container members to point directly at it, because the material array is owned by D3DX and will be destroyed as soon as this method returns.

The next section of code shows how to do everything just discussed. It stores the CTriMesh's underlying ID3DXMesh interface in the mesh container's D3DXMESHDATA member, and copies the passed D3DXMATERIAL array into a new array. A pointer to this array and the number of elements in it are also stored in the mesh container's member variables, so that they can be accessed by D3DXSaveMeshHierarchyToFile.

```
    // Store our mesh in the container
    pMeshContainer->pMesh = pNewMesh;
    pMeshContainer->D3DXMESHDATA.pMesh=pNewMesh->GetMesh();
    pMeshContainer->NumMaterials   = NumMaterials;

    // Copy over material data only if in managed mode (i.e. we can save)
    if ( NumMaterials > 0 && ManageAttribs == true )
    {
        // Allocate material array
        pMeshContainer->pMaterials = new D3DXMATERIAL[ NumMaterials ];

        // Loop through and copy
        for ( i = 0; i < NumMaterials; ++i )
        {
            pMeshContainer->pMaterials[i].MatD3D = pMaterials[i].MatD3D;
            pMeshContainer->pMaterials[i].pTextureFilename =
                     _tcsdup( pMaterials[i].pTextureFilename );

        } // Next Material

    } // End if any materials to copy
```

Now the new mesh container contains all of the information we need it to store and it is ready to be attached to the hierarchy. As our final step, we assign the D3DXMESHCONTAINER pointer passed into this function by D3DX to point to the newly allocated container. On function return, D3DX will now have access to our mesh container and will attach it to the correct frame in the hierarchy.

```
    // Store this new mesh container pointer
    *ppNewMeshContainer = (D3DXMESHCONTAINER*)pMeshContainer;

    // Success!!
    return D3D_OK;

ErrorOut:
    // If we drop here, something failed
    DestroyMeshContainer( pMeshContainer );

    if ( pDevice      ) pDevice->Release();
    if ( pAttribRemap ) delete []pAttribRemap;
    if ( pNewMesh     ) delete pNewMesh;

    // Failed....
    return hRet;
}
```

The bottom section of the code performs cleanup if any errors happen. If anything goes wrong throughout this function, program flow is diverted to this section of code and everything is correctly cleaned up.

> **NOTE**: Notice that we only make a copy of the material array and store it in the mesh container if the mesh is in managed mode. In non-managed mode, the attribute IDs of each subset may have been remapped to different values. If we try to save an actor containing a non-managed mesh, the attribute IDs would be wrong and incorrect texture and material data would be output to the file for each subset. Because of this, the CActor::SaveActorToX function only works when the actor is in managed mode. This is not a major limitation, because you will generally want to save actor data when you are using the actor as a tool (not in your actual game). In this case, using the actor in managed mode is probably perfectly acceptable from a performance perspective.

CAllocateHierarchy::DestroyFrame

The CAllocateHierarchy::DestroyFrame function is called by the D3DXFrameDestroy function when it removes the hierarchy from memory. For each frame visited by that function, this function is passed the frame that needs to be deleted. D3DX cannot de-allocate the frame for us since it has no idea how we allocated the frame memory in the CAllocateHierarchy::CreateFrame method. It also has no means of knowing what other cleanup might have to be performed.

```
HRESULT CAllocateHierarchy::DestroyFrame( LPD3DXFRAME pFrameToFree )
{
    D3DXFRAME_MATRIX * pMtxFrame = (D3DXFRAME_MATRIX*)pFrameToFree;

    // Validate Parameters
    if ( !pMtxFrame ) return D3D_OK;

    // Release data
    if ( pMtxFrame->Name ) free( pMtxFrame->Name ); // '_tcsdup' allocated.
    delete pMtxFrame;                               // 'new' allocated.

    // Success!!
    return D3D_OK;
}
```

If the passed frame pointer is valid, the memory allocated for the frame name string is deleted. Notice that we use the C free function to delete this memory because _tcsdup was used to copy the string (_tcsdup uses the C memory allocation function alloc to allocate the memory). We must always make sure we match the memory allocation functions with their proper de-allocation functions if we are to avoid memory leaks.

Once the string has been deleted, we delete the frame structure itself. Notice how we use the delete operator for this because we allocated the frame memory in CreateFrame using the C++ new operator.

CAllocateHierarchy::DestroyMeshContainer

This is the final interface function that needs to be implemented, and it is responsible for releasing the memory for a mesh container. When our actor de-allocates its frame hierarchy (using D3DXFrameDestroy), this method will be called for every mesh container. A pointer to the mesh container to be deleted is the only parameter to this function.

We begin by releasing the D3DXMATERIAL array. Recall that this array was allocated in the CreateMeshContainer method to make a copy of the passed D3DXMATERIAL array so that the D3DXSaveMeshHierarchyToFile method will be able to access the materials and textures used by the mesh during the hierarchy saving procedure. If the mesh is in non-managed mode then this array should not exist, and the mesh container's pMaterials pointer should be NULL.

```
HRESULT CAllocateHierarchy::DestroyMeshContainer( LPD3DXMESHCONTAINER
                                                  pContainerToFree )
{
    D3DXMESHCONTAINER_DERIVED * pContainer = (D3DXMESHCONTAINER_DERIVED*)pContainerToFree;

    // Validate Parameters
    if ( !pContainer ) return D3D_OK;

    // Release material data ( used for saving X file )
    if ( pContainer->pMaterials )
    {
        for ( i = 0; i < pContainer->NumMaterials; ++i )
        {
            // Release the string data
            if ( pContainer->pMaterials[i].pTextureFilename )
                free( pContainer->pMaterials[i].pTextureFilename );

        } // Next Material

        // Destroy the array
        delete []pContainer->pMaterials;
        pContainer->pMaterials = NULL;

    } // End if any material data
```

Now we release the rest of the data allocated by the mesh container. First, you will recall that we stored a copy of the CTriMesh's underlying ID3DXMesh interface in the container's D3DXMESHDATA member. We must now release this interface. Second, we must also free the memory that was allocated to store the name of the mesh. Since this memory was allocated using _tcsdup in CreateMeshContainer, we must release it using the C free function. Finally, we delete the CTriMesh object stored in the mesh container's pMesh member.

```
    // Release data
    if ( pContainer->MeshData.pMesh ) pContainer->MeshData.pMesh->Release();
    if ( pContainer->Name ) free( pContainer->Name );   // '_tcsdup' allocated.
    if ( pContainer->pMesh ) delete pContainer->pMesh;
    delete pContainer;

    // Success!!
    return D3D_OK;
}
```

We have now discussed the CAllocateHierarchy class and the CActor class in their entirety. You should be fairly comfortable with your understanding of how they work together to construct the entire frame hierarchy from an X file, taking care of frame and mesh asset allocation in the process.

Source Code Walkthrough – CAnimation

The CAnimation class is implemented in this demo to demonstrate a simple means for animating frame hierarchies. This class is not one that we will find ourselves using moving forward, but understanding its operations and the way it interacts with an actor's hierarchy should prove helpful in preparing us for the following chapter on animation.

The CAnimation class is not used by the CActor or CAllocateHierarchy classes. Objects of this type are instantiated by the scene and attached to frames in an actor's hierarchy. The scene will then use an object of this type to apply rotations via the relative matrix of the frame.

The scene will use the CActor::FindFrameByName function to search for a frame that it wishes to animate (e.g., Wheel_FL) and that function will return a pointer to the matched frame. The scene will then store a pointer to that matrix in a CAnimation object (using CAnimation::Attach). During each iteration of the game loop, the scene can then call the CAnimation::Rotate family of functions to apply rotations to the attached frame.

In our application, the scene instantiates six CAnimation objects: four will be attached to the wheels of the cars and two will be attached to the pivot frames to perform the steering animation.

The class definition (see CAnimation.h) is shown below.

```
class CAnimation
{
public:
    // Constructors & Destructors for This Class
             CAnimation();
    virtual ~CAnimation();

    // Public Functions for This Class
    void            Attach      ( LPD3DXMATRIX pMatrix, LPCTSTR strName = NULL );
    LPCTSTR         GetName     ( ) const { return m_strName; }
    LPD3DXMATRIX    GetMatrix   ( ) const { return m_pMatrix; }

    void            RotationX   ( float fRadAngle, bool bLocalAxis = true );
    void            RotationY   ( float fRadAngle, bool bLocalAxis = true );
    void            RotationZ   ( float fRadAngle, bool bLocalAxis = true );

private:
    // Private Variables for This Class
    LPD3DXMATRIX    m_pMatrix;      // Attached interpolator matrix
    LPTSTR          m_strName;      // Name (if provided) of the frame
};
```

The class only has two member variables:

LPD3DXMATRIX m_pMatrix

This member is set via the CAnimation::Attach method. It will point to the relative matrix of a frame in the hierarchy. We could use this object type without frame hierarchies, since it can serve as a generic rotation matrix generator/updater. In our demo application however, any calls to RotationX, RotationY, or RotationZ functions will apply a rotation to a frame in our automobile hierarchy.

LPSTR m_strName

This member is also set via the CAnimation::Attach method. It is the name of the animation as well as the name of the frame in the hierarchy that will have its relative matrix updated by this object.

CAnimation::CAnimation()

The constructor simply initializes the member variables to NULL.

```
CAnimation::CAnimation()
{
    // Clear any required variables
    m_pMatrix   = NULL;
    m_strName   = NULL;
}
```

CAnimation::~CAnimation()

The destructor has to take care of releasing the string memory. This memory is allocated in the Attach method using _tcsdup, so a free call is required.

```
CAnimation::~CAnimation()
{
    // Release any memory
    if ( m_strName ) free( m_strName ); // '_tcsdup' allocated.

    // Clear Variables
    m_pMatrix   = NULL;
    m_strName   = NULL;
}
```

CAnimation::Attach

This function is called by the scene to attach a matrix and frame name to the object. The passed matrix pointer is copied into the member variable first. If the string containing the name is not NULL, then it already contains a name (perhaps the object was used previously to animate another frame), so we release its memory prior to copying the passed string.

```
void CAnimation::Attach( LPD3DXMATRIX pMatrix, LPCTSTR strName /* = NULL */ )
{
    // Attach us to the matrix specified
    m_pMatrix = pMatrix;

    // Release the old name
    if ( m_strName ) free( m_strName );
```

```
    m_strName = NULL;

    // Duplicate new name if provided
    if ( strName ) m_strName = _tcsdup( strName );
}
```

CAnimation::RotationX

The application animates the attached frame via its matrix using one of three rotation functions: RotationX, RotationY or RotationZ. As all three functions are identical (with the exception that they build a rotation around a different axis), we will only show the code to the RotationX function here. Check the sourse code if you would like to see implementations for the other two.

```
void CAnimation::RotationX( float fRadAngle, bool bLocalAxis /* = true */ )
{
    D3DXMATRIX mtxRotate;

    // Validate Prerequisites
    if ( !m_pMatrix ) return;

    // Generate rotation matrix
    D3DXMatrixRotationX( &mtxRotate, fRadAngle );

    // Concatenate the two matrices
    if ( bLocalAxis )
        D3DXMatrixMultiply( m_pMatrix, &mtxRotate, m_pMatrix );
    else
        D3DXMatrixMultiply( m_pMatrix, m_pMatrix, &mtxRotate );
}
```

This function is passed an angle (in radians) describing the angle we wish to rotate the frame by. The function also accepts a Boolean parameter called bLocalAxis which allows us to toggle whether we would like the frame to be rotated about its own coordinate system axis (usually the case) or whether we would like the rotation to apply around the parent frame's axis. As the frame is relative to its parent frame in the hierarchy, this is a simple case of switching the matrix multiplication order. We will apply local rotations to our wheel frames so that they rotate around their own origins and axes.

The function creates an X axis rotation matrix using the D3DXMatrixRotationX function and then multiplies the returned matrix with the attached frame matrix. The Boolean is checked to see if local or parent relative axis rotation should be applied and multiplication order is adjusted accordingly.

We have now covered all the new objects that have been introduced in this lab project. In the next section, we will look at how they are used collectively by the scene to load a multi-mesh X file and animate the desired frames.

Source Code Walkthrough – CScene Modifications

The changes to the scene class are quite small, but they are significant. For starters, any X files loaded via CScene::LoadSceneFromX or CScene::LoadSceneFromIWF will now be loaded into CActor objects instead of directly into CTriMesh objects. In the case of an IWF file, the external references in the IWF

file will be created as actors. Even if an X file contains only a single mesh and no frame hierarchy, the CActor class can still be used in the same way. The D3DXLoadMeshHierarchyFromX function will always create a root frame in such instances and the mesh will be attached as an immediate child.

The scene object will also have its Render function updated to handle actors instead of meshes. Of course, the scene will also need to create the CAnimation objects at load time and attach them to the relevant frames in the hierarchy. The CScene::AnimateObjects function will loop through its four wheel CAnimation objects and use their member functions to apply rotations to the wheels of the car.

The updated CScene class will include a pointer to an array of CActors. This array will contain all CActors that have been loaded. While this is the only new member we have to add, we will also list the member variables that were added in the previous workbook to get a better understanding of how everything fits together. The significant member variables of CScene are shown below:

```
    // Private Variables for This Class
    TEXTURE_ITEM       *m_pTextureList [MAX_TEXTURES];  // Array of texture pointers
    D3DMATERIAL9        m_pMaterialList[MAX_MATERIALS]; // Array of material structures
    CAnimation          m_pAnimList    [MAX_ANIMATIONS];// Array of animations

    ULONG               m_nTextureCount;                // Number of textures stored
    ULONG               m_nMaterialCount;               // Number of materials stored

    CObject           **m_pObject;                      // Array of objects storing meshes
    ULONG               m_nObjectCount;                 // Number of objects

    CTriMesh          **m_pMesh;                        // Array of loaded scene meshes
    ULONG               m_nMeshCount;                   // Number of meshes
    CActor            **m_pActor;                       // Array of loaded scene actors
    ULONG               m_nActorCount;                  // Number of actors
    ATTRIBUTE_ITEM      m_pAttribCombo[MAX_ATTRIBUTES]; // Table of attribute combinations
    ULONG               m_nAttribCount;                 // Number of attributes
};
```

Let us just briefly remind ourselves of the purpose of these member variables and the data they will contain when the scene is loaded:

TEXTURE_ITEM *m_pTextureList [MAX_TEXTURES]
ULONG m_nTextureCount
This array (added in the previous workbook) will contain all textures used by the scene. Each TEXTURE_ITEM structure contains a texture pointer and a texture filename. When X files are being loaded (either via CTriMesh::LoadMeshFromX or CActor::LoadActorFromX), all meshes will have a scene texture callback function registered. The texture data (filename) from the X file will be passed to the callback function and, if the texture does not already exist, it will be loaded and added to this array. If an IWF scene is being loaded, then this array will contain the textures used by the external references (X files) and the textures stored in the IWF file that are applied to the internal meshes (non X file meshes). So this array will contain all textures used by all CActors and CTriMeshes in the scene.

D3DMATERIAL9 m_pMaterialList [MAX_MATERIALS]
ULONG m_nMaterialCount
This array is analogous to the texture array and was also added in the previous workbook. It contains all the materials used by all meshes and actors in the scene. This includes all materials loaded from X files

as well as materials used by IWF internal meshes. This array will only contain the materials used by meshes and actors created in non-managed mode. As we know, meshes in managed mode will store the material information internally in their attribute data array. This is true whether they are standalone (in their own CObject) or part of an actor's hierarchy.

ATTRIBUTE_ITEM m_pAttribCombo[MAX_ATTRIBUTES]
ULONG m_nAttribCount
This array was also introduced in the previous chapter and describes an array of texture/material combinations used by all non-managed mode meshes. It is an index into this array that is returned from the scene attribute callback, used to re-map the attribute buffer of non-managed mode meshes. Each element in this array contains the texture and material for a single scene level subset.

CObject **m_pObject
ULONG m_nObjectCount
As with all previous implementations of the CScene class that we have implemented, this array contains all the objects that need to be rendered by the scene. Now a CObject may contain a pointer to a CActor instead of a CTriMesh. If the data is loaded from an IWF file for example, this array can contain a mixture of CObject types -- some may contain standalone CTriMeshes (i.e., the meshes stored internally in the IWF file) while others contain CActors that were specified in the IWF file as external references. Each object contains a world matrix that represents the position and orientation of the object in the scene. It is this object array that the scene will loop through and render during each cycle of the game loop.

CTriMesh **m_pMesh
ULONG m_nMeshCount
This array will contain any standalone CTriMeshes that have been loaded and stored in CObjects. If the scene is loaded from an X file then this array will be empty. If the scene has been loaded from an IWF file then it will contain CTriMeshes for the meshes in the file.

CActor **m_pActor
ULONG m_nActorCount
This array will contain pointers to all CActor objects that have been loaded by the scene and stored in CObjects. These members are the only new ones introduced in this demo application. A CActor will be created by each call to the CScene::LoadSceneFromX file. Multiple actors can be created and added to this array if the CScene::LoadSceneFromIWF file is used. An actor is created for each external mesh reference stored in the file.

Now that we know how the various member variables are connected, let us cover the CScene methods that are now or are significantly modified by the introduction of CActor. The discussion of each function below will have the label 'Modified' or 'New' indicating whether the method previously existed and has just been updated or whether it is a brand new addition to the CScene class. Some methods such as the constructors and destructors have also been modified to now initialize and destroy the new CActor array. We will not show the code to these functions as they contain routine steps that we have seen dozens of times before. Check the source code for more details.

CScene::LoadSceneFromX - Modified

As we have come to expect, this method is called by the CGameApp::BuildObjects function to load X files. This function is only called if we wish a single X file to be loaded. Sometimes we may want to load IWF scenes containing multiple meshes and external X file references, and on these occasions we will use the CScene::LoadSceneFromIWF call instead. In this lab project, we are loading a single X file containing an automobile, and as such, this function will be called to start the scene construction process.

This function is not very large and is mostly unchanged from the previous version. We will look at it in two sections.

```
bool CScene::LoadSceneFromX( TCHAR * strFileName )
{
    HRESULT hRet;

    // Validate Parameters
    if (!strFileName) return false;

    // Retrieve the data path
    if ( m_strDataPath ) free( m_strDataPath );
    m_strDataPath = _tcsdup( strFileName );

    // Strip off the filename
    TCHAR * LastSlash = _tcsrchr( m_strDataPath, _T('\\') );
    if (!LastSlash) LastSlash = _tcsrchr( m_strDataPath, _T('/') );
    if (LastSlash) LastSlash[1] = _T('\0'); else m_strDataPath[0] = _T('\0');

    CActor * pNewActor = new CActor;
    if (!pNewActor) return false;

    // Load in the specified X file
    pNewActor->RegisterCallback( CActor::CALLBACK_ATTRIBUTEID, CollectAttributeID, this );
    hRet = pNewActor->LoadActorFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
    if ( FAILED(hRet) ) { delete pNewActor; return false; }
```

This function is passed the filename (complete with path) describing the name of the X file that is to be loaded. This is also the filename that will be passed and stored by the CActor that is ultimately created. The first thing the function does is make a copy of the complete filename and strip off the name of the file so that we are just left with the file path. It can often be useful for the scene object to know which folder houses its assets, so that it can be used to load any textures that the X file references.

After the path of the X file has been stored in the scene, we allocate a new CActor object. We know from our earlier discussion of CActor that it will initially contain no data and its root pointer will be set to NULL. The code then registers the CScene::CollectAttributeID static method as the CALLBACK_ATTRIBUTEID callback for the actor. This places the actor in non-managed mode, so all textures and materials contained in the X file we are about to load will be stored at the scene level. We know that this function will return global attribute IDs which are used to re-map the attribute buffers of all non-managed mode meshes.

With our non-managed mode callback registered with the actor, it is time to instruct the actor to load the X file. We call the CActor::LoadActorFromX function with the filename passed into the function and the D3DXMESH_MANAGED mesh option flag. This option flag will be stored in the actor and used to make sure that any CTriMeshes we create have their vertex and index buffers allocated in the managed resource pool. We also pass in a pointer to the Direct3D device that will own the mesh resources.

When this function returns, our CActor will have been populated with the frame/mesh hierarchy contained in the X file. Any textures and materials used by the meshes in the hierarchy will have been loaded and stored in the scene's texture, material and attribute arrays (by the callback).

Our CActor object is still a standalone object at this point, so we must add its pointer to the CScene's actor array. To do this we call a new CScene utility function called AddObject to resize the current CActor array and make room at the end for another pointer. (This function works identically to the AddMesh and AddObject utility functions from previous lessons). We then copy our new CActor pointer into the CActor array, at the end of the list. This is the list that the scene will traverse to destroy all currently loaded actors when the scene is cleaning itself up during its own destruction.

```
    // Store this new actor
    if ( AddActor( ) < 0 ) { delete pNewActor; return false; }
    m_pActor[ m_nActorCount - 1 ] = pNewActor;

    // Now build an object for this mesh (standard identity)
    CObject * pNewObject = new CObject( pNewActor );
    if ( !pNewObject ) return false;

    // Store this object
    if ( AddObject() < 0 ) { delete pNewObject; return false; }
    m_pObject[ m_nObjectCount - 1 ] = pNewObject;
```

As shown in the above code, we then allocate a new CObject. Our scene will work with CObjects so that both meshes and actors can be managed in a consistent manner. Most importantly, the CObject allows us to pair each CTriMesh or CActor with a world matrix. So when we create the CObject, we pass the CActor pointer into its constructor for storage. We then use the CScene::AddObject method to make room in the CScene's CObject array for another CObject pointer. Finally, the new CObject is added to the end of the array.

In this demo we do not use the light group mechanism from Chapter Five so that we can keep the code as focused as possible on the goals at hand. The X file will also contain no lighting information, so we will need to set up some default lights for viewing purposes. The next section of code handles this by creating four direction lights and storing them in the first four elements of the scene's D3DLIGHT9 array. These lights will be set on the device later and used by the pipeline to light our polygons. The remainder of the code is shown below:

```
    // Set up an arbitrary set of directional lights
    ZeroMemory( m_pLightList, 4 * sizeof(D3DLIGHT9));
    m_pLightList[0].Type = D3DLIGHT_DIRECTIONAL;
    m_pLightList[0].Diffuse = D3DXCOLOR( 1.0f, 0.92f, 0.82f, 0.0f );
    m_pLightList[0].Specular = D3DXCOLOR( 1.0f, 0.92f, 0.82f, 0.0f );
    m_pLightList[0].Direction = D3DXVECTOR3( 0.819f, -0.573f, 0.0f );

    m_pLightList[1].Type = D3DLIGHT_DIRECTIONAL;
```

```
    m_pLightList[1].Diffuse = D3DXCOLOR( 0.2f, 0.2f, 0.2f, 0.0f );
    m_pLightList[1].Specular = D3DXCOLOR( 0.3f, 0.3f, 0.3f, 0.0f );
    m_pLightList[1].Direction = D3DXVECTOR3( -0.819f, -0.573f, -0.0f );

    m_pLightList[2].Type = D3DLIGHT_DIRECTIONAL;
    m_pLightList[2].Diffuse = D3DXCOLOR( 0.2f, 0.2f, 0.2f, 0.0f );
    m_pLightList[2].Specular = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 0.0f );
    m_pLightList[2].Direction = D3DXVECTOR3( 0.0f, 0.707107f, 0.707107f );

    m_pLightList[3].Type = D3DLIGHT_DIRECTIONAL;
    m_pLightList[3].Diffuse = D3DXCOLOR( 0.1f, 0.05f, 0.05f, 0.0f );
    m_pLightList[3].Specular = D3DXCOLOR( 0.6f, 0.6f, 0.6f, 0.0f );
    m_pLightList[3].Direction = D3DXVECTOR3( 0.0f, -0.707107f, 0.707107f );

    // We're now using 4 lights
    m_nLightCount = 4;

    // Build Animation 'Interpolators'
    BuildAnimations();

    // Success!
    return true;
}
```

Notice at the very bottom of the above function, we make a call to the new BuildAnimations method. This function is responsible for allocating six CAnimation objects and attaching them to the wheel and pivot frames in the hierarchy. We will look at that function in just a short while.

CScene::AddActor - New

This function is included here for completeness, but you have seen code like this many times before in our studies together. AddActor resizes the actor array when one or more CActors have to be added to the scene. The function takes a Count parameter (which defaults to 1) which can allow the function to add space for multiple CActor pointers at the end of the array.

Recall that the process of resizing is simply a case of allocating a large enough array and copying over the data from the old array to the new array. The old array can then be deleted and the original pointer can then be assigned to point at the new array. This function returns the index of the first newly allocated element in the array so that the caller can start adding the new information into the array from that point.

```
long CScene::AddActor( ULONG Count /* = 1 */ )
{
    CActor ** pActorBuffer = NULL;

    // Allocate new resized array
    if (!( pActorBuffer = new CActor*[ m_nActorCount + Count ] )) return -1;

    // Existing Data?
    if ( m_pActor )
    {
        // Copy old data into new buffer
        memcpy( pActorBuffer, m_pActor, m_nActorCount * sizeof(CActor*) );

        // Release old buffer
        delete []m_pActor;
```

```
    } // End if

    // Store pointer for new buffer
    m_pActor = pActorBuffer;

    // Clear the new items
    ZeroMemory( &m_pActor[m_nActorCount], Count * sizeof(CActor*) );

    // Increase Actor Count
    m_nActorCount += Count;

    // Return first Actor
    return m_nActorCount - Count;
}
```

CScene::BuildAnimations

The CScene::BuildAnimations function is called at the bottom of both the LoadSceneFromX and the LoadSceneFromIWF functions. Its purpose is to provide the scene class with an opportunity to attach CAnimation objects to various frames stored in the actors. In our demo, this function creates six CAnimations objects and attaches them to the four wheel frames and the two pivot frames.

At the start of the function, we declare an array with six string elements. Each string in this array contains the name of the frame we wish to animate. These are the names of the frame data objects in the X file and the thus, the names of the D3DXFRAME's in the hierarchy of actors we will be searching for.

> **NOTE:** This function will not be a permanent member of our CScene class. It has been added to this lab project only to introduce you to the concept of frame animation, before we cover it in proper detail in the next chapter. Therefore, this function is almost completely hard-coded for the model we are going to use and as such, will not be very useful to you outside of this single project.

In the outer loop, we loop six times. Each iteration of this loop is searching through all frames to find a frame with the given name. For example, in the first iteration of this loop, we are searching all objects in the scene which contain actors with a frame named 'Wheel_FL'. If we find one, we attach the frame's matrix to the CAnimation object (using the CAnimation::Attach function) and we are done with this frame name. Then we continue on to the next iteration of the outer loop.

```
void CScene::BuildAnimations( )
{
    ULONG   i, j;
    LPCTSTR FrameNames[] = { _T("Wheel_FL"), _T("Wheel_FR"),
                             _T("Wheel_BL"), _T("Wheel_BR"),
                             _T("Wheel_Pivot_FL"), _T("Wheel_Pivot_FR") };

    // Search for each of the four wheels
    for ( i = 0; i < 6; ++i )
    {
        // Loop through each object and search
        for ( j = 0; j < m_nObjectCount; ++j )
        {
            CObject * pObject = m_pObject[j];
            if ( !pObject ) continue;
```

```
            // Any actor ?
            CActor * pActor = pObject->m_pActor;
            if ( !pActor ) continue;

            // Any frame by this name ?
            LPD3DXFRAME pFrame = pActor->GetFrameByName( FrameNames[i] );
            if ( pFrame )
            {
                // Attach the animation to this object, and skip to the next wheel
                m_pAnimList[i].Attach( &pFrame->TransformationMatrix, FrameNames[i] );
                break;

            } // End if found frame
            else
            {
                // Clear this interpolator out
                m_pAnimList[i].Attach( NULL );

            } // End if no frame found

        } // Next Object

    } // Next Wheel

    // If wheel yaw has already been applied, set initial rotation to this
    m_pAnimList[ Wheel_Pivot_FL ].RotationY( D3DXToRadian( fWheelYaw ) );
    m_pAnimList[ Wheel_Pivot_FR ].RotationY( D3DXToRadian( fWheelYaw ) );
}
```

If a frame cannot be found, we set the corresponding CAnimation object in the scene's array such that it is not attached to any frame. We do this by passing NULL into the Attach method.

Finally, at the bottom of this function we set the initial steering angle that has been assigned to the wheels. Feel free to set the fWheelYaw variable to an initial angle other than zero if you want the automobile wheels rotated by some amount. Notice that, as discussed earlier, we apply a Y axis rotation to the pivot frames so that both of the car's front wheels are rotated around the car's up vector. Because the CAnimation objects are already attached to their frames at this point, we can simply use their RotationY methods (for the CAnimation objects attached to the pivots) to perform this task.

One thing of interest in the above function is that we are accessing to the two pivot animation objects using Wheel_Pivot_FL and Wheel_Pivot_FR as array indices. These values, as well as the indices for the other frames, are defined in the scene module's namespace, as shown below:

```
namespace
{
    const UCHAR AuthorID[5]       = { 'G', 'I', 'L', 'E', 'S' };
    const UCHAR Wheel_FL          = 0;
    const UCHAR Wheel_FR          = 1;
    const UCHAR Wheel_BL          = 2;
    const UCHAR Wheel_BR          = 3;
    const UCHAR Wheel_Pivot_FL    = 4;
    const UCHAR Wheel_Pivot_FR    = 5;
    float       fWheelYaw         = 0.0f;
    float       fWheelYawVelocity = 1000.0f;
};
```

Until now, only the AuthorID has existed in the namespace -- it was used by the IWF loading function to identify GILES™ custom chunks. Now we have added six Wheel_ members which describe the indices of the CAnimation objects in the scene that use the specified frames. We also have three new members at the bottom. The variable fWheelYaw will contain the current angle at which the pivot frames are rotated (i.e., the angle at which the front wheels are rotated). We saw this variable used in the previous code segment to set the rotation of the wheels. The variable fWheelYawVelocity contains the speed at which the front wheels will rotate left and right. We will see this variable used in a moment, when we examine the CScene::ProcessInput function.

CScene::AnimateObjects - Modified

This function is called by CGameApp::FrameAdvance for each cycle of the main game loop. This function has been a part of this class from the very start and provides the scene a chance to update its object during each frame.

In this version of the function, we rotate the four wheel frames via their matrices so that the car's wheels are continuously revolving around their local X axes. This function rotates the wheels at two revolutions per second (i.e., 720 degrees per second). We calculate the rotation angle for a given call by scaling 360 degrees by the number of revolutions (2) and then multiplying this by the elapsed time since the last frame. This will usually be fractions of a second, scaling the rotation angle for any particular update to only a few degrees. We pass the result of this calculation into the D3DXToRadian function so that we have the rotation angle in radians (required by the CAnimation::Rotation functions).

```
void CScene::AnimateObjects( CTimer & Timer )
{
    float fRevsPerSecond = 2.0f, fRadAngle;

    // Generate wheel frame rotation angle.
    fRadAngle = D3DXToRadian( (360.0f * fRevsPerSecond) * Timer.GetTimeElapsed() );

    // Rotate the wheels
    m_pAnimList[ Wheel_FL ].RotationX( fRadAngle );
    m_pAnimList[ Wheel_FR ].RotationX( fRadAngle );
    m_pAnimList[ Wheel_BL ].RotationX( fRadAngle );
    m_pAnimList[ Wheel_BR ].RotationX( fRadAngle );

}
```

Notice that because we have set up array index variables in the module namespace, we can quickly index into the CScene::m_pAnimList array and call the RotationX function for the correct CAnimation objects.

CScene::ProcessInput – Modified

CScene::ProcessInput is called by CGameApp::ProcessInput, which itself is called by CGameApp::FrameAdvance. The actual trapping of key data and subsequent camera updates are found in the CGameApp:ProcessInput function, as we saw in all previous lessons. When it calls this function,

it passes in an array of all the keys with their up/down state. In this function we are testing for the comma key and the period key, which allow the user to rotate the wheels left or right respectively.

The first thing this function does is check the VK_COMMA and VK_DOT indices of the key array. If either has its top four bits set, then they key is depressed and rotations need to be applied. To provide a smooth deceleration of the wheel when the user lets go of the key, we use the fWheelYawVelocity variable to store the turn velocity. We can think of this value as the number of degrees per second that we wish the wheels to turn left or right. Every time the key is down during this function call, another 400 degrees per second is added to the turn velocity. The velocity can be either positive or negative for left and right steering. If no keys are pressed, then 400 degrees per second is subtracted from the velocity each time the function is called. This allows the wheel rotations a more realistic ramping up/down when the keys are pressed and released.

```
void CScene::ProcessInput( UCHAR pKeyBuffer[], CTimer & Timer )
{
    float fAngle = 0.0f;

    // Apply rotations ?
    if ( pKeyBuffer[ VK_COMMA ] & 0xF0 )
        fWheelYawVelocity -= 400.0f * Timer.GetTimeElapsed();
    else if ( pKeyBuffer[ VK_DOT ] & 0xF0 )
        fWheelYawVelocity += 400.0f * Timer.GetTimeElapsed();
    else
    {
        // Decelerate the yaw
        if ( fWheelYawVelocity > 0 )
            fWheelYawVelocity -= min(fWheelYawVelocity, 400.0f * Timer.GetTimeElapsed() );
        else
            fWheelYawVelocity -= max(fWheelYawVelocity, -400.0f * Timer.GetTimeElapsed());
    } // End if declerate
```

At this point the yaw velocity is scaled by the elapsed time to give us the actual number of degrees we need to rotate the wheels in this frame. Remember, the rotations are being added incrementally. The fWheelYaw member contains the current turn angle of the wheels -- it has the incremental turn angles added to it each time the function is called. We clamp the turn angle to 45 degrees, so if the yaw angle exceeds this amount with the newly calculated incremental angle added to it, we set the yaw angle to –45 or +45 and recalculate the incremental angle to stay within these bounds.

Once the incremental angle has been calculated and clamped, we pass it into the CAnimation::RotationY method of the animation objects attached to both pivots. Since the pivots are defined in the root's frame of reference, a rotation around the pivot Y axis rotates the wheel about the automobile's up vector.

```
    // Get the angle increase for this frame
    fAngle = fWheelYawVelocity * Timer.GetTimeElapsed();

    // Anything applicable
    if ( fAngle != 0.0f )
    {
        // Apply to total yaw and clamp where appropriate
        fWheelYaw += fAngle;
        if ( fWheelYaw >  45.0f )
           { fAngle -= (fWheelYaw - 45.0f); fWheelYaw = 45.0f; fWheelYawVelocity = 0.0f; }
```

```
        if ( fWheelYaw < -45.0f )
           { fAngle -= (fWheelYaw + 45.0f); fWheelYaw = -45.0f; fWheelYawVelocity = 0.0f; }

        m_pAnimList[ Wheel_Pivot_FL ].RotationY( D3DXToRadian( fAngle ) );
        m_pAnimList[ Wheel_Pivot_FR ].RotationY( D3DXToRadian( fAngle ) );

    } // End if any rotation
}
```

If you are a little rusty with this kind of velocity scaling, refer back to Chapter Four where we used a similar technique for our camera movement system.

CScene::Render

The render function is responsible for setting the states and rendering the subsets of all CObjects in the scene. The CObjects can contain both actors and/or meshes, so we must make sure that we take this into account. The reason the scene is responsible for state setting in this application is that we created our actors/meshes in non-managed mode. Most of the following code should be familiar to:

```
void CScene::Render( CCamera & Camera )
{
    ULONG i, j;
    long  MaterialIndex, TextureIndex;

    if ( !m_pD3DDevice ) return;

    // Setup some states...
    m_pD3DDevice->SetRenderState( D3DRS_NORMALIZENORMALS, TRUE );

    // Render the skybox first !
    RenderSkyBox( Camera );

    // Enable lights
    for ( i = 0; i < m_nLightCount; ++i )
    {
        m_pD3DDevice->SetLight( i, &m_pLightList[i] );
        m_pD3DDevice->LightEnable( i, TRUE );

    } // Next Light
```

The function first checks that the device is valid and sets the D3DRS_NORMALIZENORMALS render state to true. This will make sure that even if a mesh or actor world matrix includes a scaling characteristic that would ordinarily scale the normals, no problems with the lighting pipeline will be encountered.

We then render call the RenderSkyBox function to render the skybox mesh (if one exists). This will only be the case if the scene has been loaded from an IWF file and the file contains a skybox entity.

The final section of the above code sets up the lights on the device and enables them. Unlike earlier demos where the light group system was employed, in this demo we are simply setting up the first N lights found in the IWF file. Alternatively, if we are loading the scene straight from an X file, four direction lights are hard-coded and set up as we saw earlier.

Why are we setting the same lights each time the render loop executes if they never change? The reasons are twofold. First, if you decide to integrate the light group system into this demo yourself, you will absolutely need to set and unset the lights belonging to each light group before rendering its assigned polygons. Second, this approach allows you to place code in the scene class that might dynamically change the light properties. By setting the light properties each time, we make sure that any dynamic changes to a light slot's properties will be reflected in the next render.

The next section of code contains the significant changes from previous implementation of CScene. We render our scene by looping through each CObject in the scene object array. For each object we check its CTriMesh pointer and its CActor pointer to see which are valid. This will tell us whether the object contains a single mesh or an actor encapsulating a complete mesh hierarchy. If the mesh pointer is valid then we set the object's world matrix and FVF flags on the device. If the actor pointer is valid then we pass the object's world matrix into the CActor::SetWorldMatrix function. Earlier we saw that by passing true for the second parameter (as in this case) the hierarchy will be traversed and the absolute world matrices for each frame in the hierarchy will be generated. At this point, either the mesh or the actor is ready to be rendered.

```
    // Process each object
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CActor    * pActor = m_pObject[i]->m_pActor;
        CTriMesh * pMesh  = m_pObject[i]->m_pMesh;
        if ( !pMesh && !pActor ) continue;

        // Set up transforms
        if ( pMesh )
        {
            // Setup the per-mesh / object details
            m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i]->m_mtxWorld );
            m_pD3DDevice->SetFVF( pMesh->GetFVF() );

        } // End if mesh
        else
        {
            // Set up actor / object details (Make sure we update the frame matices!)
            pActor->SetWorldMatrix( &m_pObject[i]->m_mtxWorld, true );

        } // End if actor
```

Because the scene is using its meshes and actors in non-managed mode, we must now loop through every global subset in the scene's ATTRIBUTE_ITEM array, fetch the texture and material for that subset, and set them on the device. If a subset contains no material, then a default material of bright white is used. If a subset has no texture, then the texture stage is set to NULL and the subset will be rendered in shaded mode only.

With the states set, we call either CTriMesh::DrawSubset or CActor::DrawSubset to render the given subset depending on which type of object we are processing.

```
        // Loop through each scene owned attribute
        for ( j = 0; j < m_nAttribCount; j++ )
        {
            // Retrieve indices
```

```
            MaterialIndex = m_pAttribCombo[j].MaterialIndex;
            TextureIndex  = m_pAttribCombo[j].TextureIndex;

            // Set the states
            if ( MaterialIndex >= 0 )
                m_pD3DDevice->SetMaterial( &m_pMaterialList[ MaterialIndex ] );
            else
                m_pD3DDevice->SetMaterial( &m_DefaultMaterial );

            if ( TextureIndex >= 0 )
                m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ]->Texture );
            else
                m_pD3DDevice->SetTexture( 0, NULL );

            // Render all faces with this attribute ID
            if ( pMesh )
                pMesh->DrawSubset( j );
            else
                pActor->DrawActorSubset( j );

        } // Next Attribute

    } // Next Object

    // Disable lights again (to prevent problems later)
    for ( i = 0; i < m_nLightCount; ++i ) m_pD3DDevice->LightEnable( i, FALSE );

}
```

As we know, CTriMesh::DrawSubset simply wraps a call to the ID3DXMesh::DrawSubset function, instructing the underlying mesh to render its subsets. If the object contains an actor then CActor::DrawSubset is called and this function does a lot more work. We examined this function earlier and saw how it traverses every frame in the hierarchy and searches for frames that have mesh containers attached. For each one found, the frame's world matrix is set as the device world matrix and then CTriMesh::DrawSubset is called for each mesh attached to that frame. This will cause all matching subsets in all meshes contained in the hierarchy to be rendered.

Finally, we disable all of the lights at the end of the function. While this is not strictly necessary in our current demo, it does allow us to safely animate or even delete lights from the scene's light array if we wanted to. This step makes sure that if a scene's light is deleted, that its properties will not continue to be used by the light slot is was previously assigned to.

CScene::ProcessReference

Before wrapping up this workbook, we will examine the modified CScene::ProcessReference function, which now handles creating actors from IWF file external X file references.

As usual, the scene uses the IWF SDK classes to load data from IWF files and then calls various functions (ProcessMeshes, ProcessMaterials, ProcessVertices, etc.) to extract the data from the IWF SDK objects and copy it into its own arrays. Recall that the ProcessEntities function is called to process any entities stored in the IWF file and this function in turn called the ProcessReference function if an entity being processed was a GILES™ reference entity. The ProcessReference function was

implemented in the previous workbook to extract the filename of an external reference and use it to load a CTriMesh and add it to the scene. Now, this function (while mostly unchanged) creates a CActor from the reference filename instead.

We will not dwell long on this function since much of the actor creation and setup is the same as the CScene::LoadSceneFromX function which we have already covered.

The first section of the function checks that this is an external reference before continuing. It then builds the filename string used to create the absolute filename (with path) for the X file being loaded. The IWF ReferenceEntity object only contains the name of the X file and does not include path information (for obvious reasons). We do this by adding the entity name to the path name stored in the scene's m_strDataPath string. This data path was stored in this string in the LoadSceneFromIWF function by truncating the absolute filename passed in.

```
bool CScene::ProcessReference( const ReferenceEntity& Reference,
                               const D3DXMATRIX & mtxWorld )
{
    HRESULT   hRet;
    CActor  * pReferenceActor = NULL;

    // Skip if this is anything other than an external reference.
    // Internal references are not supported in this demo.
    if (Reference.ReferenceType != 1) return true;

    // Build filename string
    TCHAR Buffer[MAX_PATH];
    _tcscpy( Buffer, m_strDataPath );
    _tcscat( Buffer, Reference.ReferenceName );
```

We will not immediately load the actor (like we do in the LoadSceneFromX function) because the IWF file might contain multiple instances of the same actor in the scene. So, first a search is done through the scene's CActor array to see if an actor exists with a matching name. If one is found then we will use this pointer, otherwise, we will need to create a new CActor, register the scene callbacks and load the file:

```
    // Search to see if this X file has already been loaded
    for ( ULONG i = 0; i < m_nActorCount; ++i )
    {
        if (!m_pActor[i]) continue;
        if ( _tcsicmp( Buffer, m_pActor[i]->GetActorName() ) == 0 ) break;

    } // Next Actor

    // If we didn't reach then end, this Actor already exists
    if ( i != m_nActorCount )
    {
        // Store reference Actor.
        pReferenceActor = m_pActor[i];
    }   // End if Actor already exists
    else
    {
        // Allocate a new Actor for this reference
        CActor * pNewActor = new CActor;
        if (!pNewActor) return false;

        // Load in the externally referenced X File
```

```
        pNewActor->RegisterCallback(CActor::CALLBACK_ATTRIBUTEID,CollectAttributeID,this );
        hRet = pNewActor->LoadActorFromX( Buffer, D3DXMESH_MANAGED, m_pD3DDevice );
        if ( FAILED(hRet) ) { delete pNewActor; return false; }

        // Store this new Actor
        if ( AddActor( ) < 0 ) { delete pNewActor; return false; }
        m_pActor[ m_nActorCount - 1 ] = pNewActor;

        // Store as object reference Actor
        pReferenceActor = pNewActor;

    } // End if Actor doesn't exist.
```

At this point, the local variable pReferenceActor is pointing to the actor we need. This will either be a new actor we have just loaded or a pointer to an actor that already existed in the scene's CActor array.

Our final task is to add a new CObject to the scene's object array and store this actor pointer in it. As it happens, IWF references also contain a world matrix describing where the reference is situated in the scene, so we can copy this matrix from the reference object into the CObject as well.

```
    // Now build an object for this Actor (standard identity)
    CObject * pNewObject = new CObject( pReferenceActor );
    if ( !pNewObject ) return false;

    // Copy over the specified matrix
    pNewObject->m_mtxWorld = mtxWorld;

    // Store this object
    if ( AddObject() < 0 ) { delete pNewObject; return false; }
    m_pObject[ m_nObjectCount - 1 ] = pNewObject;

    // Success!!
    return true;
}
```

Conclusion

In this lab project we put in place the foundations of a very useful class called CActor, which we will see used repeatedly throughout the rest of this course. This will be true in next chapter when we cover hierarchical animation and in the following chapters when we cover skinning and skeletal animation. The CActor class is a very welcome addition to our toolkit and it will make our job a lot simpler from here on in.

While we lightly touched on hierarchical animation in this chapter, the next chapter will be devoted entirely to this subject. We will learn how to use the D3DX animation controller to load complex keyframed animations that can be blended with other animations and applied to our hierarchy. These techniques will then be carried into later lessons where the animation controller will be used to animate the skeletal system of game characters and other dynamic entity types. Be sure that you understand how our hierarchy (and CActor) system works before moving on, since it is the core of just about everything we will do in the next few lessons.

Chapter Ten

Animation

Introduction

To date our coursework has focused primarily on the three dimensions of physical space. We looked at how to build, position, and orient models in a 3D world so that we can take a snapshot of the scene and present it to our player. But there is also a fourth important dimension that we have looked at only briefly: time. Time is the dimension introduced when animation becomes a part of our game design.

To animate something means literally to "give life" to it. This is clearly an important concept in computer games. Of course, most of our lab projects have included animation in one form or another. From the spinning cubes back in Chapter One to the movement of our player/camera through the world as the result of user input, we have had a degree of animation in almost all of our demos. In this lesson we will look at some of the more traditional animation concepts that one thinks about when working with 3D computer graphics.

In today's 3D computer games, being able to animate 3D models and scenes in a controlled or pre-recorded way is absolutely essential. We will see later in the course how animated characters in computer games will have many pre-recorded animations that our application can select and playback at runtime in response to some game event. For example, a character might include pre-recorded animations for walking and running so that as it travels from point A to point B in the world, it looks correct while doing so. There may also exist pre-recorded animations of several death sequences for the character that can be played back by our application in response to the character's health falling below zero. We would also need animations to play when character itself is firing its weapon, etc.

Most of the difficult work of building and capturing animation data is the domain of the project artist, modeller, and animator. Most popular 3D modelling packages include tools to animate 3D models and export that data. As programmers, our job will focus on integrating the results of their efforts into our game engine. Our only real requirement is that the data be delivered in a format supported by the engine we have designed. In this particular lesson, that format will be the X file format. Fortunately, most commercial modelling programs support the export of animated scenes to the X format, either natively or through the use of a plug-in. Those of you who are fortunate to own or have access to high-end programs like 3D Studio Max™ and Maya™ will indeed have a great opportunity to design and export complex animations. There are also excellent packages like Milkshape3D™ that are a more cost-effective solution for the hobbyist, that still offer a wealth of animation and export options.

> **Note:** Our focus in this lesson will be on working with pre-recorded animation sequences that are created offline by our project artist. However, it should be noted that there are also animation techniques that can be calculated procedurally at runtime to produce certain interesting behaviours. These techniques (such as inverse kinematics to cite just one example) will be explored later in this course series.

Real-time animation is such an integral part of today's videogame experience. Not so many years ago, video games would typically play pre-recorded movie clips called 'cut-scenes' at key points in the game to help move the storyline forward. While the quality of those cut-scenes generally exceeded the in-game graphics quality, there were a number of criticisms that followed from their use. First, they would often be associated with large load times which would destroy the sense of immersion and game flow.

Perhaps more importantly, the player was transformed from being a participant in a 3D world where they had total freedom of movement, to passively watching non-interactive slide shows or movies.

In modern games, given that the hardware they run on has more dedicated processing power for graphics and animation, cut-scenes are commonly played out using the actual game engine. This allows them to become an integral part of the game experience, where in-game story events can happen on the fly without the need to cut away and make the player idly watch. Player immersion is preserved and the world feels much more real. Players can often continue to move about the world even while these scripted events are playing out. If you load in Lab Project 10.1 you will see a very simple pre-scripted animation of a spaceship taking off and flying into the distance. While this is the simplest of examples as far as such things go (normally such events are triggered as part of an evolving storyline), the point is that you are still allowed to move freely around the world while the sequence plays out around you. Imagine that just prior to the ship taking off, your squad mate informed you that he would commandeer the ship and meet you at the rendezvous point. Then you watched him jog off into the distance under heavy fire, board the ship, and take off for deep space. All the while, you were laying down covering fire for him as he made his way across the dangerous killing field. While our demo is not nearly as exciting, you can certainly imagine such a sequence taking place in a commercial game. Modern 3D game titles are littered with these small but often critical sets of scripts and animations and they are vital to making the 3D world feel more realistic. But even when this is not the case and a game actually cuts away and the player must simply passively watch the sequence, using the in-game graphics engine provides a sense of consistency that keeps the player immersed in the world.

Finally, it is worth noting that it is much more efficient to store scripts and animations for the game engine (the animations are generally built from simple translation and rotation instructions that can be interpolated -- more on this shortly) than it is to store full motion video sequences at 30 frames per second. Such videos can consume significant disk space, and require important processor time especially if used frequently.

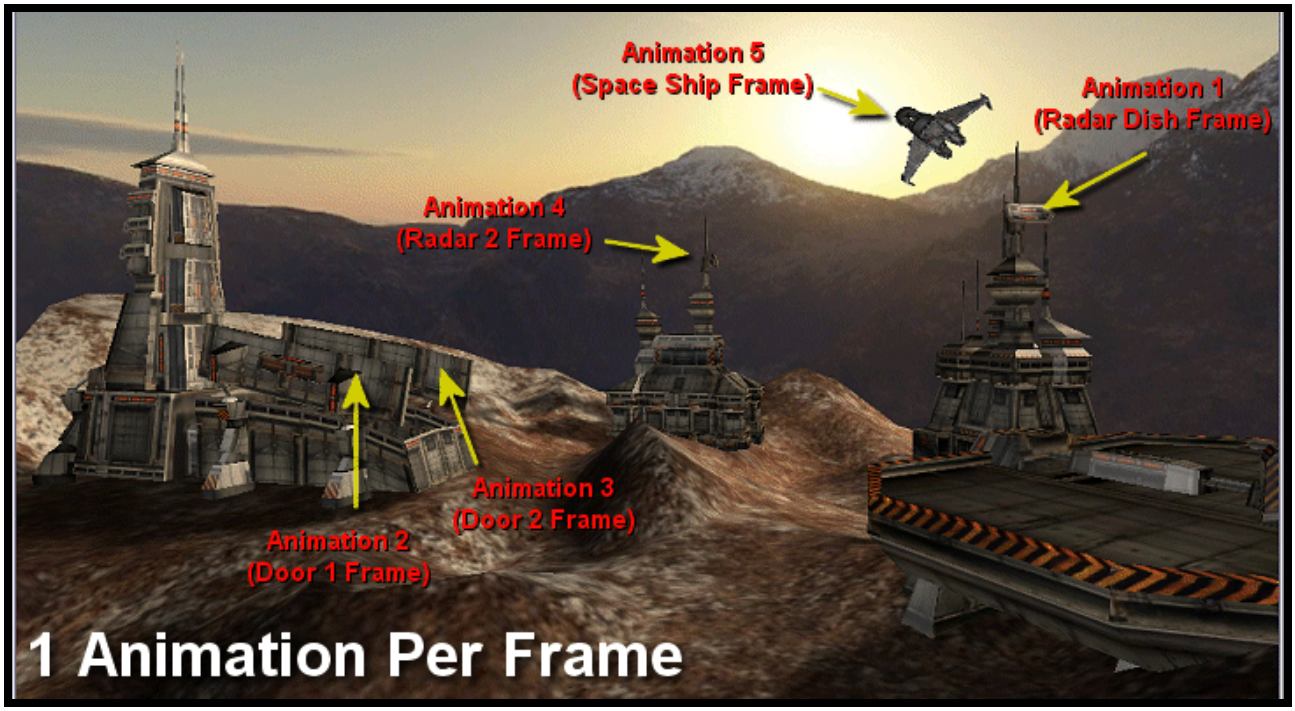
10.1 Basic Animation

Before we examine the underlying mechanisms of the DirectX animation system, it will be encouraging to take a quick look at how easy it is to load a hierarchical X file that includes animation data and then play that animation in our game engine. D3DXLoadMeshHierarchyFromX allows us to load hierarchical X files that include animation, so we are already on the right track. The final parameter we pass to the function is the address of a pointer to an ID3DXAnimationController. If the file contains animation data, then this pointer will ultimately be used to access, manipulate, and play those stored sequences. If the file contains no animation data, then this pointer will be set to NULL on function return. With this in mind, let us see how we would load an animated X file. In many ways this next code snippet brings together everything that we have learned so far:

```
CAllocator Allocator;
D3DXFRAME * pRootFrame = NULL;
ID3DXAnimationController * pAnimController = NULL;
D3DXLoadMeshHierarchyFromX( “MyScene.x”, D3DXMESH_MANAGED, pD3DDevice, &Allocator,
                            NULL, &pFrameRoot, &pAnimController );
```

The animation data stored in the X file will be used to manipulate the values stored in the hierarchy frame matrices. When loaded by D3DX, each frame in the hierarchy that the artist decides to animate will have attached an **Animation Data Set** that contains a set of animation information for a single frame in the hierarchy. You should recall our earlier theoretical discussion about frame animation in the Chapter Nine -- our CAnimation object simply applied a rotation to the frame to which it was attached. But we are already familiar with the concept of these objects being attached to frames in the hierarchy and being responsible for managing their animation. While D3DX does not create a separate object to contain the animation data for a single frame in the hierarchy, we can think of the animation data for each frame as being a packet of animation data that is assigned to an animation set and registered with the animation controller. For example, if the file contains a single animation set that animated 10 frames in the hierarchy, D3DX would create an animation set object that contained 10 packets of animation data, one for each frame. Each packet is also assigned a name that matches the name of the frame to which it is assigned and to which its animation data applies. This name is also referred to as the name of the Animation. Therefore, in DirectX terminology, when we discuss an 'Animation' we are referring to a set of animation information linked to a single frame in the hierarchy and contained inside an animation set.

Lab Project 10.1 provides a basic demonstration of animating a scene hierarchy. It includes a small animated spacecraft, a pair of hanger doors that open and close, and rotating radar domes on two of the buildings. The entire scene is stored as a single hierarchy, but only certain frames include attached animations. There are in fact, five animations in total: one for each animated frame in the scene. Static frames do not have animations created for them. The five animations in this scene were all created by D3DXLoadMeshHierarchyFromX automatically when we loaded the X file. An animation set was created and the five animations were added to it. The animation set was then registered with the animation controller so that on function return, our application can start playing the animation(s) immediately.



At a high level, the animation controller returned from the D3DXLoadMeshHierarchyFromX function is like an **Animation Set Manager**; it stores all of the animations found in the X file (in one or more animation sets) and manages communication between them and the application. When the application wants to advance the global timeline of the animated scene, we instruct the animation controller via its AdvanceTime method. The animation controller in turn relays this timeline update to each of its currently active animation sets. Each animation set is a collection of frame animations. Each frame animation stored in the animation set is used to generate the new matrix for the frame to which it is attached for that position in the timeline. In short, with a single call to the ID3DXAnimationController::AdvanceTime function, all active animation sets update the frame matrices for which they have animations defined.

> **Note:** While the first part of this lesson will focus on populating the D3DX animation system with animation data stored in X files, we will see later that D3DX provides the ability to construct all the frame interpolators, animation sets and the animation controller manually. This would allow an application to import data from any file format into the D3DX animation system. This would be done by manually attaching animations to the various frames in the hierarchy and registering them with animation sets on the animation controller. We will discuss how to manually build the components of the D3DX animation system later.

While the techniques for creating simple animated scenes vary from application to application, one concept remains universal: **the animation timeline**. Every animation package includes a timeline where transformations or other types of events can be placed in chronological order. These events are called *keyframes* and they represent the most important transitions that must take place at particular moments in time. Time itself can be measured in seconds or milliseconds or by the more arbitrary timing method of 'ticks'. When D3DX loads animation sets from an X file, every animation set will include its own animation timeline. Therefore, while our application will be advancing a single global time via the ID3DXAnimationController::AdvanceTime method, each animation set has its own local timeline which the animation controller takes care of updating. The length of an animation set's timeline (also referred to as its *period*) is equal to the length of its largest defined **animation**. Remember that an **animation** is a set of animation data for a single frame in the hierarchy.

> **Note:** As the D3DX loading functions load their animation data from the X file format, it is important that you install the appropriate 'X file Exporter Plug-in' for your chosen 3D modelling application. 3D Studio Max™ is a very popular choice for artists and modellers, so if this is your chosen application then you are in luck since there are a couple to choose from. While the DirectX9 SDK ships with an 'X File Exporter' plug-in for 3D Studio Max™, you are provided with only the source code. This means you must compile the plug-in yourself, which comes with its own set of problems as discussed in Chapter 8. Luckily, there are some very good 3rd party X file exporters for MAX (one called 'Panda X File Exporter' which is freely available is worth investigating). There are also exporter plug-ins available for other popular programs, such as Maya™ and Milkshape™ for example.

Each animation set will have been defined by the artist using its own local time. As several animation sets may be playing at once, each of which may have different durations (periods), the ID3DXAnimationController::AdvanceTime is used to calculate and mix the local animation timelines using a single global time. For example, Animation Set 1 may have a duration of 10 seconds, while Animation Set 2 may have a duration of 30 seconds. If both of the animation sets were created to be looping animations (more on this in a moment), then their timelines will wrap back around again to the beginning when the global time of the animation controller exceeds their local time. If the global time of the animation controller was at 35 seconds, at which periodic position on their local timelines would

each of the above animation sets be? Animation Set 1 would be 5 seconds into its local timeline and Animation Set 2 would also be 4 seconds in. Why? Consider the wrapping property of the animation sets. At 35 seconds, Animation Set 1 (which only lasts of 10 seconds) would have wrapped around three times and would now be on the fifth second of its fourth payback. Animation Set 2 which lasts for 30 seconds reached the end of its timeline 5 seconds ago, so now has a local periodic position of 5 seconds into its second playback.

Updating the animation sets and instructing them to rebuild their associated frame matrices is actually quite simple. The ID3DXAnimationController interface includes a method called AdvanceTime which serves as the main update function for the animation controller and all its underlying animation sets. The application will pass in a value in seconds that represents the *elapsed* time (i.e., the time since the last call to AdvanceTime) that we would like to use to progress the animation controller's global timeline. This value is passed on to each animation set in the scene so that their local timelines are updated, and wrapped if necessary. After an animation set timeline update, the animation set will loop through each of its stored animations and use their data to re-build the frame matrices assigned to them from the hierarchy.

The following code shows a simple game loop that cycles the animation:

```
void GameLoop( float ElapsedTime )
{
      pAnimController->AdvanceTime( ElapsedTime );
      RenderFrame( pRootFrame, NULL );
}
```

The GameLoop function would be called every frame and is passed the amount of time (in seconds or fractions of a second) that has passed since the last render. The call to ID3DXAnimationController::AdvanceTime updates the controller's global time and also passes the elapsed time along to each of its animation sets so that their local timelines are also advanced. Each animation set will use this new local time to perform a look up in its list of stored animations. Each animation (which stores animation keyframes for a single frame in the hierarchy) will be used to calculate what the matrix for the assigned frame should look like at the specified time. When the function returns, all of the animations stored in the animation set will have updated the matrices of their attached frames in the hierarchy.

> **Note:** It is important to emphasize that time drives the animation loop. This is critical to maintaining smooth animation independent of frame rate. While it might seem easier to simply increment a frame counter for controlling animation, it is much more risky given the different speeds your game will run on different hardware. Locking the frame rate can minimize the problems, but that is not usually a desirable method as it imposes a limitation you probably do not want. Nor can you count on all hardware even meeting your minimums. Since the animation sequences in modelling packages are based on time, the engine should be as well. This is certainly easy enough to accommodate, so do not try to cut corners here.

Now all we have to do is render the hierarchy. As we saw earlier, this involves stepping through the hierarchy, combining the relative matrices as we go. Any of these relative frame matrices that have had animations assigned (in an active animation set) will have been updated at this point, so the changes will automatically affect all child frames and meshes. For example, if we were to apply a rotation animation

to the root frame, although there would be only a single animation in the animation set currently being played, and the AdvanceTime method would only affect one matrix directly (the root matrix), the entire frame hierarchy will rotate because all child matrices are relative transformations. When we build the world matrices for each frame during the traversal and matrix concatenation process (i.e., the update pass), these changes would be reflected down through all children. So essentially, by advancing the global animation time via the animation controller, we are simply instructing all contained animations to update their attached frame matrices to reflect what the position and orientation of the frame should be at this time. Remember, it is the relative frame matrices that are updated -- the ones that were loaded from the X file. We still have to generate the world matrices for each frame using the standard traversal and concatenation technique, after these relative matrices have been updated.

As discussed earlier, it is often preferable to separate our rendering logic into an update pass and a drawing pass. Recall that this involved maintaining a local copy of the world matrix for each frame that is updated as the hierarchy is traversed. We can modify our previous GameLoop function to accommodate this strategy very quickly:

```
void GameLoop( float ElapsedTime )
{
      pAnimController->AdvanceTime( ElapsedTime );
      UpdateHierarchy( pRootFrame, NULL );
      RenderHierarchy( pRootFrame, pCamera );
}
```

It is worth noting that the example GameLoop function above does not bother wrapping around the global animation time value but simply continues to advance it. As it happens, if the animation sets have been created such that they should loop, it will automatically take any value that is out of bounds and turn it into a time within its animation set's timeline based on the length of the animation set. For example, if we have an animation set that runs for 15 seconds (i.e., its longest animation runs for 15 seconds) and the global time of the animation was currently at 18 seconds, it would be treated as 3 seconds into the second animation loop. A value of 31 would be treated as 1 second into the third loop of the animation, and so on. If your animation sets are not configured to loop then you might not want to continue increasing the global time of the animation controller. Instead you might want to reset it back to zero when it gets too large. This will allow the non-looping animations to play again when the global time restarts at zero.

The DirectX animation system underwent significant change in the 2003 summer update of the SDK. Previously, individual animations (formerly called Interpolators) within an animation set also had their own local timelines. This third time layer was extremely useful as it meant that you could have several animations within an animation set, all with different durations and independent looping capability. This was arguably more convenient, but perhaps this third layer of individual animation independence was slowing down the entire system. This could potentially justify its demise. However, because individual animations within an animation set are now just 'dumb' animation data containers for frames in the hierarchy, the artist will often have to make sure that all animations assigned to a given animation set last for the same duration. This is because the animation set's local timeline only wraps around when the end of its period is reached (its period is the duration of its longest running animation). When the 2003 summer update was released, the assets for Lab Project 10.1 had to be completely rebuilt. Initially, the five animations in our space ship scene were all part of the same animation set. The radar dish

animations only lasted a short period of time compared to the length of time it took the spaceship to take off and fly away into the distance. We can see from watching the scene that the radar dishes will rotate many times during this sequence. Previously, it did not matter that the radar dish animation did not last as long as the space ship animation because although they belonged to the same animation set, the animations themselves had their own loop-capable timelines. Once the radar dish had finished one complete cycle, it would just wrap around and do it again. This support has now been removed from DirectX 9 and there is no per-animation timeline concept anymore. To upgrade our assets, we had to insert extra animation data for the radar dishes (adding extra rotations implicitly) so that the duration of every animation lasted as long as the flying ship. All animations now have the same duration, which is equal to the duration of the animation set to which the animations belong. One might argue that this was not exactly a change for the better, but so it goes.

So at its simplest, rendering an animated X file scene involves only one additional function call (ID3DXAnimationController::AdvanceTime). Very often, this may be all your application needs to do. Of course, the D3DX animation subsystem is much more sophisticated than this and includes many additional interfaces and function calls. But we can see now that including pre-generated animations in our existing game engine is actually quite an easy thing to do.

The remainder of our chapter will cover the D3DX animation interfaces and the functionality they expose. We will soon see that the ID3DXAnimationController has much more depth than we saw in this section. While it can be used in the very simple way we just learned, where the entire scene was animated using just one function call, it can also be used to produce animation techniques of great complexity.

10.2 Keyframe Interpolation

Animation is a time-driven concept. We know that each frame in our hierarchy can be assigned its own animation which maintains a timeline of animation information for that frame. Along the timeline at specified intervals are keyframes which store transformation information such as position, rotation, and scale. This is a very efficient way to store animation. By interpolating the transformation data stored in the keyframes at runtime, we are able to dynamically generate our animation data (a frame matrix) for any given moment in the game. This means that we will not need to store a copy of the scene data in every possible position/orientation along the timeline; we simply need a handful of stored keyframes that represent the important ‘poses’ along the timeline. This saves a lot of storage space, and certainly makes the artist’s life much easier. Animation data is stored in an X file (and in most modelling files) as keyframes.

> **Note:** Understanding how keyframe animation works is simpler if you know how to create the data in a modelling package. A small tutorial on how to create animation data in a 3D modelling package can be found in Appendix A. It would be helpful if you read this short tutorial before continuing.

Let us now spend a little time discussing keyframes in more detail and how the D3DX animation controller will use them to generate animation data at runtime.

One of the earliest forms of animation was the flipbook. A flipbook contains a collection of pages, each with a drawing that is slightly altered from the previous page, such that when we rapidly flip the pages from front to back or vice versa, the drawing appears to animate.

Let us imagine a 100 page flipbook containing an animation of a ball rolling along the ground from left to right. On page 1 the ball is at its furthest point left, and on page 100 the ball has been drawn at its far right position. For all pages in between, the ball is drawn such that it moves further to the right as the page number increases.

If someone were to ask you to guess the position of the ball on an arbitrary page in the book, your guess would probably be reasonably correct. On page 50, you would know that it should be equidistant from its start and end positions. On page 25, it would be about 25% of the way from its start position with about 75% of the path left to traverse. We can make these assumptions because we know in advance that the animation is designed to represent a linear translation between two known locations over a known number of pages (assuming a constant speed of course).

Now imagine that we rip out all of the pages except the first and last pages. Do you think that you could fill in the images on the 98 missing pages by hand? Of course you could. This is because you know where the ball needs to be on any given page.

Keyframes are essentially like the first and last pages in the flipbook. They tell us the state of an object at a given point in an animation sequence. Once we know the starting state and ending state, it just becomes a case of filling in the gaps.

Let us think of this example in the context of our animation controller. Our hierarchy will contain a single frame (call it BallFrame) with an attached animation. The animation would store two keyframes. The first keyframe would describe the starting position of the ball frame and the time at which the ball should be placed at this position. Since this is the start keyframe, the time will be zero. The second keyframe would contain the end position for the ball and the time at which the ball should reach this position. Let us say for example that animation will last 20 seconds. So keyframe 2 will have a timestamp of 20.

> **Note:** This is a simple example for the purposes of explaining the concept. A keyframe's timestamp is not normally stored explicitly as seconds, but is usually stored using a measurement of time with a higher granularity. This can actually be an arbitrary time unit as we will see in a moment. Either way, D3DX will take care of loading this data in so that we can specify our time in seconds in the 'AdvanceTime' method of the ID3DXAnimationController interface.

Now, let us say we were to call ID3DXAnimationController::AdvanceTime with a value that would advance the global timeline of the controller to 15 seconds. Let us also assume that the animation set's local timeline is also set at 15 seconds. For each animation contained within an animation set, we can locate the two keyframes that most closely bound the passed time (i.e. the closest keyframes to the left and right of the current time on the timeline). In this simple example, we will say there are only two keyframes describing a position at 0 seconds and a position at 20 seconds. These are the start and end positions of our 'rolling ball' flipbook analogy. We know that in this case the two keyframes that are closest to either side of the passed time (15 secs) will be the start keyframe (0 secs) and the end keyframe (20 secs).

With the bounding frames located, the animation set will now use the passed time to linearly interpolate between the translation values stored in the keyframes. This will determine exactly where the ball should be at 15 seconds. Next we see a very simple example of what the animation set might do to calculate the position of an animation called 'Ball_Animation' at 15 seconds. For reasons of clarity we are referencing the animation data for 'Ball_Animation' as if it is a C++ object. In reality, it would just be an array of keyframes stored in the animation set assigned the name 'Ball_Frame'.

```
PassedTime = 15.0;

D3DXVECTOR3 StartPos = Ball_Animation.KeyFrame[0].Position;  // Position at 0 seconds
D3DXVECTOR3 EndPos   = Ball_Animation.KeyFrame[1].Position;  // Position at 20 seconds

float s  =  PassedTime – Ball_Animation.KeyFrame[0].Time;
      s /= ( Ball_Animation.KeyFrame[1].Time – Ball_Animation.KeyFrame[0]Time);

CurrentPosition = StartPos + (s * (EndPos - StartPos));
```

In reality, when the current position is calculated, it would be placed inside the frame matrix that the ball animation is attached to. We do not have to implement any of this code, as it is all handled by the D3DXAnimationSet object contained inside the animation controller. We are currently just trying to get a better feel for how the whole system works.

What if we wanted the ball in our example to jump up in the air somewhere around the middle of the animation? Not a problem. More complex animations simply require more key-frames, every subsequent pair of which describes a single transition between events. To create the jump effect, we could use the keyframes seen in Fig 9.6.

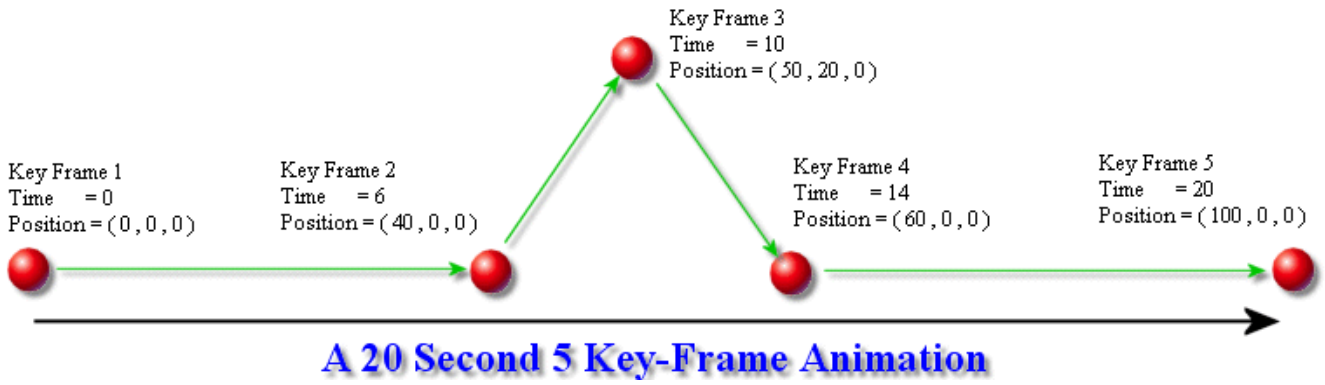


Figure 9.6

Using the example in Fig 9.6, let us think about what will happen if we passed in an elapsed time of 8 seconds.

First the animation set would find the two bounding keyframes for this time. Since we passed in a time of 8 seconds, the two bounding keyframes would be keyframe 2 and keyframe 3 at 6 and 10 seconds respectively. We would then use the passed time to linearly interpolate between the two positions.

```
D3DXVECTOR3 KeyFrameAPos  = ( 40 , 0 , 0 );   // Low Closest Key Frame '2'
float       KeyFrameATime = 6;

D3DXVECTOR3 KeyFrameBPos  = ( 50 , 20 , 0 );  // Higher Closest Key Frame '3'
float       KeyFrameBTime = 10;

float       PassedTime  = 8 ;       // The time out application requested

float s =  PassedTime – KeyFrameATime;       // = 2
      s /= (KeyFrameBTime – KeyFrameATime);  // = 2 / ( 10-6 ) = 0.5

Current Position  = KeyFrameAPos + ( s    *  (KeyFrameBPos – KeyFrameAPos );

            // = ( 40 , 0 , 0 )  + ( 0.5  *  ( 10 , 20 , 0 ) );
            // = ( 40 , 0 , 0 )  + ( 5 , 10 , 0 );
            // = ( 45, 10 ,0 )
```

The final position calculated would place our ball exactly half way along the green direction vector between keyframes 2 and 3 in the diagram.

So far we have looked only at translation-based animation examples, but often, animations will contain rotation and scaling sequences as well. The concepts are identical in these cases -- scaling will also use a linear interpolation approach (just like we saw with translation key frames) while rotations (stored as quaternion keyframes) will use a spherical linear interpolation (i.e. slerp) to accomplish the same objective. While these are the only keyframe types that D3DX supports in its animation controller, you could certainly use keyframes to manage other types of events in your engine. In Module III of this course series for example, we will process a set of keyframes that store color and transparency events for animating particles.

D3DXAnimationSets can store their keyframe data for a given animation in two storage styles. The first style uses separate lists for the individual transformation types (translation, rotation, or scaling). This is referred to as the SRT storage model (**S**cale**R**otate**T**ranslate). There is a list of scale vectors, a separate list of translation vectors and a list of rotation quaternions describing all the scale, translation and rotation keys assigned to that single animation respectively (the SRT for a single hierarchy frame). Every instruction in each list is assigned the time at which the instruction should be applied to the attached frame. Keyframe interpolation has to be done for each list in isolation (for times in between keyframes) to generate a final scale, position and orientation which are used to update the attached frame matrix. This is referred to as *SRT interpolation*. In the animation controller, an animation is really just a list of keyframes (usually in SRT format) and a frame name describing the name of the frame to which the keyframes are applied.

The second storage style uses a single list of 4x4 matrices that represent the combination of all three transform types. In this format, each animation is stored as a single list of matrix keys along with their assigned times. When this time is reached in the animation set's local timeline, the corresponding matrix will become the frame matrix to which it is attached. For times in between keyframes, the two matrices that most closely bound the desired time will be used to interpolate a new matrix that will be used instead.

While it may seem more convenient to store animations as a single list of matrix keys, there are times when storing matrix keyframes is less than ideal (we will discuss this later). As programmers, the choice of whether each animation's keyframes are stored as matrix keys or as separate scale, rotation and translation keys (SRT Keys) is usually out of our hands. We have to adapt our approach according to how that data is stored in the X file. The modeler or artist who created the X file may well have a choice when exporting their data to X file format as to which keyframe data format they prefer, so if you have preference for your engine, it is best to let them know in advance.

Once we know how the keyframe animation data is stored in an X file, things will start to make a lot more sense. Let us explore that concept in the next section.

10.3 X Files and Animation

Animation data is stored in an X file using three standard templates. We will look at each in turn.

The first template we will study is the **Animation** template, which manages the keyframe data for a single frame in the hierarchy. When the D3DX loading function encounters an Animation data object, it will create and add a new animation to the animation set currently being constructed. The animation itself (which is just an array of keyframes) will be populated by the keyframes specified in the Animation data object in the X file. Therefore, the Animation data object in an X file is analogous to an 'Animation' in the D3DX animation system. The X file Animation data object will contain one or more **AnimationKey** child data objects that store the actual keyframe data. There will also be a child data reference object which is the name of the frame in the hierarchy for which this keyframe data applies.

Using our automobile example from the last chapter, let us assume that our artist has applied rotation data to the wheels. In this case there would be four Animation data objects in the file -- one for each wheel frame in the hierarchy. D3DX would create an animation set with four animations. Each animation would be attached to a wheel frame.

The Animation template is an open template defined in DirectX as follows:

```
template Animation
{
      <3D82AB4F-62DA-11cf-AB39-0020AF71E433>
      [...]
}
```

The first thing an object of this type will need is the name of the frame in the hierarchy that it animates. For example, in our bouncing ball example we might give the frame that stores the ball the name 'Ball_Frame'. We would embed this name in the Animation object in the X file as follows:

```
Animation BallAnimation
{
      { Ball_Frame }
}
```

Now that the Animation object is linked to a frame in our hierarchy, we will need to add some keyframe data. If the animation has been saved in SRT format (scale, rotation, translation) then there could potentially be three AnimationKey data objects that exist as children of the Animation object.

Each AnimationKey data object will contain a list of one or more keyframes for a specific key type. For example, the first AnimationKey object may contain a list of Scale keyframes, where each keyframe describes the scale for the attached frame matrix at a particular point in the timeline. The second AnimationKey object might contain a list of Rotation keyframes where each keyframe in this list describes the orientation of the attached frame at a given point along the timeline. Finally, the third AnimationKey object might contain a list of Translation keyframes describing the position of the attached frame at key points in the animation timeline. In the SRT format we can see that all three transformation types are stored separately, although this is not necessarily a one-to-one mapping. For example, we may have 10 scale keyframes, 150 rotation keyframes, and 0 translation keyframes. In this example the Animation object would contain only two child AnimationKey objects because there is no translation data to be stored. If the animation data is stored in matrix format instead of SRT format, the Animation data object will always contain only a single child AnimationKey data object that acts as a container for the animation matrix keys.

The AnimationKey template is a standard X file template and is defined as follows:

```
template AnimationKey
{
      <10DD46A8-775B-11cf-8F52-0040333594A3>
      DWORD keyType;
      DWORD nKeys;
      array TimedFloatKeys keys[nKeys];
}
```

DWORD KeyType

The AnimationKey object can store individual transformation components or a combined matrix. This first member identifies the type of keyframe data contained in the object. The possible values are shown in the table below.

KeyType	Description
0	The AnimationKey object will contain a list of rotation keyframes. Each key frame stores a rotation quaternion in a 4D vector representing the parent relative orientation of the frame.
1	The AnimationKey object will contain a list of scale keyframes. Each keyframe will store a 3D vector describing how to scale the frame along its X, Y, and Z axes respectively.
2	The AnimationKey object will contain a list of translation vectors. Each keyframe in the list will be represented as a 3D vector describing the position of the frame relative to its parent frame.
4	The AnimationKey object will contain a list of matrix keyframes. Each keyframe will be a 4x4 matrix containing the relative scale, rotation, and translation information for the frame at the assigned time

> **Note:** The SDK 9.0 documentation states that a value of '3' denotes the AnimationKey as containing matrices. This is incorrect and the correct value to identify an AnimationKey as a matrix keyframe container is '4' as shown in the above table.

DWORD nKeys

The second member of the AnimationKey template describes the number of keyframes in the list that follows. For example, if the KeyType member was set to 2 and the nKeys member was set to 22, this would indicate that the animation key contains 22 translation keyframes.

array TimedFloatKeys keys[nKeys]

The final member of the AnimationKey template is an array of TimedFloatKeys, which are the keyframe data. Each element in this array is a single keyframe and the array will be large enough to hold as many keyframes as are described in the nKeys member.

The TimedFloatKey type is another standard template and is defined as follows:

```
template TimedFloatKeys
{
    < F406B180-7B3B-11cf-8F52-0040333594A3 >
    DWORD time;
    FloatKeys tfkeys;
}
```

An object of this type will exist for each keyframe in the array. The first member contains the time where this keyframe can be found on the timeline. We will discuss time units (seconds, milliseconds, etc.) shortly. The FloatKeys child object is another standard template object:

```
template FloatKeys
{
   < 10DD46A9-775B-11cf-8F52-0040333594A3 >
   DWORD nValues;
   array float values[nValues];
 }
```

The FloatKeys object allows us to store arbitrary length vectors. Rotation keyframes are represented as 4 floats (a quaternion), while scaling and translation keyframes require only 3 floats. We can also store matrix keyframes as a 4x4 matrix (16 floats). The nValues member will tell us how many floats are stored in the array.

TimedFloatKeys data objects are really just wrappers around FloatKeys data objects with an additional time component.

With all of this in mind, let us examine a simple X file with two animated frames. The first frame (MyFrame1) will include rotation and translation animations, but no scaling. This means the Animation object for this frame will have two AnimationKey child objects -- one containing a list of rotation keyframes, the other a list of translation keyframes.

The second frame (MyFrame2) will use a list of matrix keyframes instead of separate scale, rotation and translation lists. As a result, the Animation object for this frame will contain a single AnimationKey child data object with a list of matrix keyframes.

We will look only at the relevant part of the X file so you can just assume that the frame hierarchy and meshes are stored elsewhere.

```
Animation   Anim1     // Animation Data for 1st animated frame
{
      { MyFrame1 }

      AnimationKey
      {
            0;    // Rotation Key Frames
            4;    // Four rotation key frames in list

            //    Time  NumFloats   Quaternion Data
            0;    4;    1.00000 ;  0.00000 ; 0.00000 ; 0.70000;;;  // Key Frame 1
            5;    4;    0.22222 ;  0.00000 ; 0.03333 ; 0.20000;;;  // Key Frame 2
            10;   4;    0.02000 ;  0.00000 ; 1.00000 ; 0.70000;;;  // Key Frame 3
            20;   4;    0.22222 ;  1.00000 ; 0.03333 ; 0.20000;;;  // Key Frame 4
      }

      AnimationKey
      {
            2;    // Translation Key Frames
            3;    // 3 Translation key frames in list

            //    Time  NumFloats   3D Translation Vectors
            0;    3;     0.00000 ; 0.00000 ; 0.70000;;; // Key Frame 1
            2;    3;     0.00000 ; 4.03333 ; 0.20000;;; // Key Frame 2
            4;    3;     0.00000 ; 1.00000 ; 3.70000;;; // Key Frame 3
      }
}

Animation Anim2    // Animation data for 2nd animated frame
{
      {MyFrame2}

       AnimationKey
       {
            4;    // This Animation Key holds Matrix Key Frame Data
            3;    // Three Key Frames

            //    Time  NumFloats   Matrix Data
            0;    16;   1.00000;    0.00000;    0.00000;    0.00000; // Key 1
                        0.00000;    1.00000;    0.00000;    0.00000;
                        0.00000;    0.00000;    1.00000;    0.00000;
                        0.00000;    0.00000;    0.00000;    1.00000;;;
            20;   16;   1.00000;    0.00000;    0.00000;    0.00000; // Key 2
                        0.00000;    1.00000;    0.00000;    0.00000;
                        0.00000;    0.00000;    1.00000;    0.00000;
                        10.00000;   0.00000;    0.00000;    1.00000;;;
            30;   16;   1.00000;    0.00000;    0.00000;    0.00000; // Key 3
                        0.00000;    1.00000;    0.00000;    0.00000;
```

```
                        0.00000;    0.00000;    1.00000;    0.00000;
                        20.00000;   30.00000;   0.00000;    1.00000;;;
      }
}
```

Notice that there are three semi-colons at the end of each keyframe. This is because, in addition to ending the float (one semi-colon), we are also closing the FloatKey object (another semi-colon) and the outer TimedFloatKey object (the third semi-colon). For more information on why these semi-colons appear, please read the section in the DX9 SDK docs entitled "Use of Commas and Semicolons".

> **Note:** Quaternions are stored inside X files with their components in WXYZ order, not the more typical XYZW format of 4D vectors.

It is important to emphasize that D3DXLoadMeshHierarchyFromX will automatically load the keyframe data (regardless of the format it is in) and internally store the data for each of its animations inside an animation set in **SRT format**.

The next template we will look at is called AnimationOptions. This is another potential child object of the Animation data object (the X file version of a 'frame animation') and is part of the standard template collection. This object will tell us whether this frame animation is a looping animation or not as well as whether interpolation between keys should be linear or spline based (splines are curves and will be discussed later in the series). The template used to describe the animation options is:

```
template AnimationOptions
{
  < E2BF56C0-840F-11cf-8F52-0040333594A3 >
    DWORD openclosed;
    DWORD positionquality;
}
```

The first DWORD indicates whether the animation is open or closed. If we set this value to '0' then the animation is described as a 'closed' animation and it will not loop. If we set this value to '1' then the animation is 'open' and should loop.
The second member describes whether linear interpolation or spline-based interpolation should be performed between keyframes. A value of 0 requests cheaper but lower quality linear position interpolations, while a value of one requests smoother but more expensive spline-based position interpolations.

> **Note:** Based on some internal testing results, it would appear that for the moment at least, this template type is not properly recognized by the D3DX animation controller (DX 9.0b). Please keep this in mind if you choose to use this template in your own X files.

The next animation related X file template that we will examine is the AnimationSet:

```
template AnimationSet
{
      <3D82AB50-62DA-11cf-AB39-0020AF71E433>
      [Animation]
}
```

The AnimationSet data object is a container for Animation objects (note that it is a restricted template). All Animation objects will be contained within a parent AnimationSet. In the previous chapter on hierarchies, you will remember that we actually wrote our own CAnimationSet class that was basically a container for CAnimation objects. The relationship between the X file AnimationSet data object and the X file Animation data object is analogous to the relationship between our proprietary CAnimationSet class and the CAnimation class. However, as mentioned, the data for a single frame animation is not stored in its own objects but stored directly inside the animation set to which it is assigned (likely in a large array of animations). The animation set is also more than just a container; it also provides the logic to interpolate between the SRT keyframes stored in each of its animations to generate the frame matrix for each animation at a specific periodic position in its timeline. The animation set also has its own local timeline, although that is managed at the animation controller level by the animation mixer (more on this later).

We will see later in this chapter that we can actually have multiple AnimationSets assigned to the same hierarchy. This is especially useful when animating a character hierarchy, where one animation set would contain all of the Animations (i.e., frame animations) to make the character walk while a second animation set would contain the Animations to make the character fire a weapon. Both animation sets would reference the same hierarchy, and indeed the Animations in two different animation sets will often animate the same hierarchy frame. Since the D3DX animation controller allows us to mix multiple animation sets together, we can even simultaneously blend the walking animation with the weapon firing animation so that a character can fire while on the move. We will return to explore this concept in great detail later in the lesson.

Next we see how animation data might look in a stripped-down X file using the animation set concept just discussed.

```
AnimationSet Walk // Walking Animation
{
      Animation   // Animation data for 1st animated frame
      {
            {MyFrame1}
            AnimationKey
            {
                  4;    // This Animation Key holds Matrix Key Frame Data
                  3;    // Three Key Frames
                  Key-Frame Matrix Data Goes here
            }
      }

      Animation   // Animation data for 2nd animated frame
      {
            {MyFrame2}
            AnimationKey
            {
                  4;    // This Animation Key holds Matrix Key Frame Data
                  3;    // Three Key Frames
                  Key-Frame Matrix Data Goes here
            }
      }
}  // end Walk animation set
```

```
AnimationSet Shoot     // Shooting animation
{
      Animation   // Animation data for 1st  animated frame
      {
            {MyFrame1}
             AnimationKey
             {
                  4;    // This Animation Key holds Matrix Key Frame Data
                  3;    // Three Key Frames
                  Key-Frame Matrix Data Goes here
             }
      }

      Animation   // Animation data for 3rd animated frame
      {
            {MyFrame3}
             AnimationKey
             {
                  4;    // This Animation Key holds Matrix Key Frame Data
                  3;    // Three Key Frames
                  Key-Frame Matrix Data Goes here
             }
      }
}  // end 'Shoot' animation set
```

Note that you do not have to give an animation set a name since they can be referenced by index once they have been loaded by D3DX. However, giving them a name is always a good idea if given the choice in your X file exporter. The D3DXLoadMeshHierarchyFromX would load the sample X file and would create two animation sets that would then be registered with the animation controller. Each animation set in this example would have two animations.

In the walk animation set above, Frame1 and Frame2 of the hierarchy are manipulated. For the shoot animation, Frame1 and Frame3 are manipulated. If these animation sets were blended together, then Frame1, Frame2, and Frame3 would be animated simultaneously. Frame1 would be influenced by both animation sets because there would be an animation in both animation sets attached to it and updating its matrix.

Unfortunately, at least at the time of this writing, many X file exporters are able to save only a single animation set in the X file. This is true regardless of how many logical types of animations are created for the scene (walk, run, jump, etc.). To be clear, the frames for these logical animations are all saved; they just happen to be stored in the same animation set in many cases. While you can usually configure the modeller to export only a specific range of keyframes, often the entire hierarchy will need to be saved as well. However even if you can avoid this, or code around it, or simply do not care about the extra space, it is still much easier to manage your animation data if it is not all stored in a single set.

On the bright side, in Lab Project 10.2 we will create a small tool to circumvent this problem. Our GUI tool will allow the user to load an X file that contains all of its animations in a single animation set and then use the D3DX animation interfaces to break those animations into separate named animation sets before resaving the file. Apart from being a very handy tool, it will also get us acquainted with using the D3DX animation interfaces to build an ID3DXAnimationController and ID3DXAnimationSet objects

from scratch. This is something you will definitely want to learn how to do if your animation data is stored in a file format other than X.

10.3.1 Timing and X Files

As it turns out, the timing values stored within the keyframes of an X file are not specified in seconds; time is instead specified using *ticks*. By itself, this timestamp format is fairly useless, for it is largely arbitrary, and depends on factors such as how fast we want the animation to run and how many of these ticks the exporting application (the 3D modelling program) was processing per second. For D3DX to process these keyframe timestamps in a meaningful way and convert them into seconds, there is an additional X file template, called `AnimTicksPerSecond`, which can be placed anywhere in the file. It contains a single DWORD which tells us (and D3DXLoadMeshHierarchyFromX) how many of these ticks equal one second in application time. The D3DX loading function uses this data to convert keyframe timestamp values stored in the X file into seconds for storage within its interpolator objects.

```
template AnimTicksPerSecond
{
      DWORD NumberOfTicks;
}
```

As it turns out, this data is omitted from most X files. If it is not included, D3DX assumes a default of 4800 ticks per second. This is the default also assumed by most 3D modelling applications which export to the X file format. This means that if the last keyframe in any given animation has a timestamp of 67200, the animation will have a length of 14 seconds. Although this may seem to be an unnecessary middle level of interpretation introduced at load time, it does allow for the easy alteration of an animation's speed and length by simply changing the number of ticks per second value stored in the X file. Perhaps we might find that the animation data exported from a certain application runs too fast and we would like to slow it down. Without this Ticks Per Second concept, we would have to manually adjust the timestamps of every keyframe in the X file (a laborious process indeed).

There are a few traditional `AnimTicksPerSecond` values that many developers/modellers use beyond the default of 4800. Values like 30, 60, 24, and 25 are often used as well. These values generally correspond to 'frames per second' standards that are used across the industry. If we were to use the 30 ticks per second case as an example, this would mean that for a keyframe to be triggered ten seconds into the animation, its timestamp should be 300. Again, this is all handled on our behalf by the D3DX animation controller and its animation sets, so it may not seem relevant. However, it certainly helps to understand how the timing is interpreted when building animations manually, or when writing your own exporter.

Before moving on to the next section, let us quickly sum up what we know so far. First, we now have a good understanding of how animation data is represented in an X file. We know that there will usually be a single animation set, and it will contain one or more animation data objects. Each animation data object is connected to a single frame (by reference) in the frame hierarchy (contained elsewhere in the X file) and stores the keyframe data for that frame, essentially describing that single animation's timeline. We know that inside the animation object, the keyframe data can be stored in SRT format, where there

will be three separate keyframe lists (three AnimationKey objects) for scale, rotation, and translation data. Alternatively, the data can be stored as a single AnimationKey using a keyframe list of matrices storing the combined scale, rotation, and translation information. We also know that regardless of how the data is stored in the X file, once loaded, the animation data will be stored inside the animation set as SRT lists for each animation defined. We learned that the timestamp of each keyframe in an animation is defined in the X file using an arbitrary unit of measurement called a tick, which is given meaning by the exporter by either inserting an AnimTicksPerSecond data object somewhere in the file or by using the D3DX default (4800 ticks/sec). Finally, we know that more often than not, we will let D3DX do all of the hard work for us. We will simply call ID3DXAnimationController::AdvanceTime to generate/update the relative matrices for each frame to correspond to the elapsed time for the currently active animation(s).

10.4 The D3DX Animation Interfaces (Overview)

Now that we know how our animation data is stored at the file level, let us examine how it is arranged in memory once D3DX has loaded it using D3DXLoadMeshHierarchyFromX. As it turns out, there is not much difference between the two cases.

When D3DXLoadMeshHierarchyFromX loads in animation data, that data is stored and managed by a D3DX animation subsystem consisting of two different COM object types. This exposes two main animation interfaces to our application (there are others used for intermediate tasks). We will look at each of them very briefly here first, and then in more detail as the chapter progresses.

10.4.1. The Animation Controller

The top level of the animation subsystem is the ID3DXAnimationController interface. An interface to this object is returned to our application by the D3DXLoadMeshHierarchyFromX function. It is this interface that our application will work with most frequently. We can think of it as being an animation set manager -- when animation sets are loaded from the X file, each animation set is registered with the newly created animation controller and an interface to this controller is returned to the application.

The animation controller also provides a very powerful *animation mixer* that can be used to assign animation sets to different ‘tracks’ and blend them together when the animations are played. The animation mixer can be thought of as being somewhat like a multiple-track audio mixing desk, where we would feed in individual pieces of audio data (drums, guitar, vocals, etc.) on different ‘tracks’ or ‘channels’ and blend them together based on the properties that we have set for each track (ex. volume, bass, treble, etc.). What comes out of the mixing desk output jack is all of the audio data, from all of the tracks, combined together into a final product that can be saved to CD and played back.

If we needed to play only a single animation set and did not require blending, the animations sets would all still be loaded and registered with the animation controller by the D3DXLoadMeshHierarchyFromX function; we would just assign the animation set we wish to play to track 1 and play it back by itself.

Note: Even if an X file contains multiple animation sets, by default D3DXLoadMeshHierarchyFromX will assign the only last animation set defined in the file to track 1 of the mixer. No other mixer tracks or animation sets will be set up for you, so if you wish to play them, you will have to assign these animation sets to the mixer tracks manually. This is why you will not see us setting up the animation mixer or doing any kind of animation set initialisation in Lab Project 10.1. The X file only contains a single animation set and it is assigned to track 1 by default. When the loading function returns, we are ready to start playing the animation set immediately.

10.4.2. The Animation Set

The ID3DXAnimationSet interface and its underlying COM object mirrors the X file AnimationSet template almost exactly. This is the base class interface and it is actually the ID3DXKeyframedAnimationSet that we will be using most of the time. There is also an ID3DXCompressedAnimationSet which is an animation set containing compressed keyframe data. Once keyframe animation data has been loaded by D3DX into an ID3DXKeyframedAnimationSet, we can use the methods of this interface to create a compressed animation set if we wish.

The base interface (ID3DXAnimationSet) is pure abstract, thus allowing you to derive your own animation set classes which interpolate between the keyframes of each of its animations in a customized manner. As long as the derived class exposes the functions of the base interface, the animation controller will happily work with it to generate the frame matrices without caring how that matrix data was generated.

For each AnimationSet data object contained in the X file, D3DXLoadMeshHierarchyFromX will create a new D3DXKeyframedAnimationSet object and register it with the animation controller. On function return, we can use the methods of the ID3DXAnimationController interface to assign these animation sets to tracks on the mixer so that they are ready for playback. Each animation set will contain the SRT data of each animation data object stored in the file. It will also contain a mapping describing which keyframe animation data sets are assigned to which frame(s) in the hierarchy. This is a simple name match. In other words, the name of an animation within an animation set matches the name of its assigned frame in the hierarchy.

Recall that an X file animation object is essentially a container of AnimationKey data objects for a single frame in the hierarchy. This same relationship is maintained in D3DX. In this case however, each Animation Set will contain the array of SRT keyframes along with a frame name. It is the SRT keyframe set for a single frame in the hierarchy which is referred to as being a ***frame animation***, or an ‘Animation’ for short. The contents of each Animation inside of an animation set are equivalent to the X file AnimationKey data object contents. For each frame that is animated in the hierarchy, a matching animation will be created and attached to that frame and managed by D3DX via its parent animation set. The Animation stores all the keyframe data for that frame for the animation set to which it is assigned.

For example, if an X file contains an animation set that animates five frames in the hierarchy, a D3DXKeyframedAnimationSet object will be created and registered with the animation controller, and five Animations will be created and stored inside it. They will contain the individual keyframes for each

of the animated hierarchy frames. These animations are owned by the animation set and the animation set will be owned by the animation controller.

We will hardly ever need to work with the ID3DXAnimationSet interface or the ID3DXKeyframedAnimationSet interface when playing back an animation. Often, our application's only exposure to these interfaces will occur when we wish to use the animation mixer to blend animation sets, or assign animation sets to mixer tracks. For example, ID3DXAnimationController has a function called SetTrackAnimationSet which takes two parameters: a track number and an ID3DXAnimationSet interface. Calling this method will assign the passed animation set to the specified track on the animation controller's mixer. When using the animation controller in this simple way, we will often have no need to actually call any of the animation set's methods. The methods of the animation set are usually called by the animation controller to generate the SRT data for each animated frame in the hierarchy (in response to a call to AdvanceTime from the application).

Keyframe animation sets are much more than simple animation containers. In fact, when our application calls ID3DXAnimationController::AdvanceTime, the animation sets manage much of the total animation process. They select the appropriate keyframes for a given moment in time for each animation and then use them to generate dynamic animation data that transforms the frames to which the animations are attached.

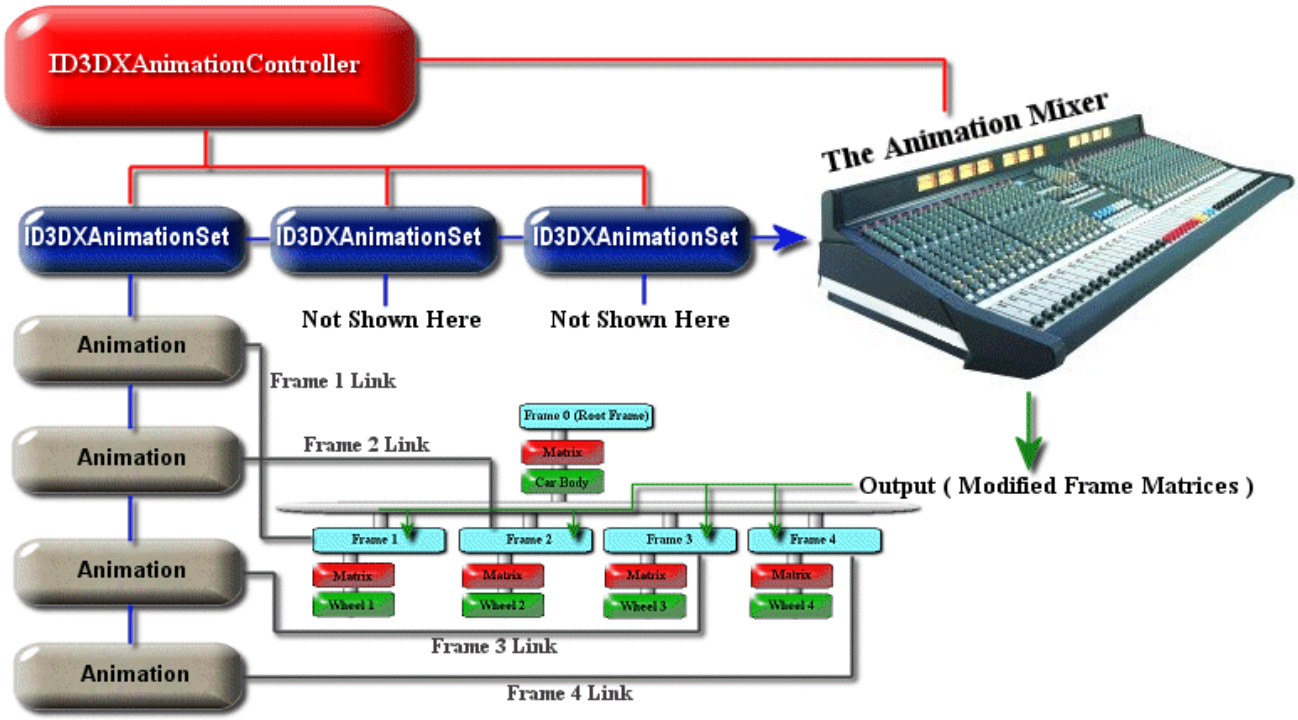
When ID3DXAnimationController::AdvanceTime is called, the animation controller will begin by looping through each of its currently active AnimationSets (an *active* set is one that has been assigned to a track on the animation mixer). For each animation set, it will use the ID3DXAnimatonSet interface to fetch each of its underlying animations. For each animation, the controller will call the ID3DXAnimationSet::GetSRT function. GetSRT will use the current animation local time (i.e., periodic position) of the animation set to search for its two bounding keyframes and then interpolate a new value somewhere in between. The output is a new position, scale and rotation for the frame to which it is attached. The animation controller will do this for each animation contained in each animation set. At the end of this process, the animation controller will contain the new matrix information for each animated frame in the hierarchy. These may be placed directly into the frame matrices or, depending on how the mixer is configured, this matrix data might be blended together to create the final matrix data for the assigned frame in the hierarchy. So, in the case of the ID3DXKeyframedAnimationSet interface, the animation set is responsible for performing the actual keyframe interpolation and handing the results back up to the animation controller for mixing.

The animation controller will typically cache the new frame position, scale, and rotation information generated by the ID3DXAnimationSet::GetSRT function call (for each animation) until all active animation sets have been processed. Then the new frame data for each animation set will be scaled by the weight assigned to its track. If two animation sets have keyframe animations defined for the same frame in the hierarchy, then the results of each animation will be blended together taking the track weights into account. The results are then stored in the parent-relative frame matrix for the appropriate owner frame in the hierarchy.

> **Note:** We can also use the ID3DXKeyFrameAnimationSet interface to directly work with the keyframe data. This can be useful if you are designing your own animation tool and need the ability to create, edit, or destroy keyframe data values. For example, we can use the ID3DXKeyframedAnimationSet::RegisterAnimationSRTKeys method to place our own keyframes directly

into an animation set that we manually create. This allows us to register animations manually with the animation set. This would be useful if you wish to use the D3DX animation system but intend to load the animation data from a custom file format. You could programmatically build the animation controller, create and register animation sets with the controller, and then add you own animations to the animation set. The keyframe data for each animation you add could be the data imported from your custom file format.

Fig 9.7 depicts an animation controller that owns three animation sets. You will notice that we are reusing our automobile hierarchy once again and that for the first animation set, there is an animation for each wheel frame (the grey lines show the frame that each animation is connected to). When we call ID3DXAnimationController::AdvanceTime, the clocks for each track on the mixer are incremented by this value. Once the timers for each track have been updated, the current time of the track is mapped to the local time of the animation set assigned to it (to account for looping, etc). Each animation set uses this time to calculate a new position, rotation, and scale for each animation by interpolating its stored keyframes (via its GetSRT function). It passes the results back up to the animation controller where they are fed into the animation mixer. The result of the mixing process is a new frame matrix generated using any or all of the animation sets results. This data is stored in the frame matrices so that the next time we update and render the hierarchy, the matrices have been adjusted and the children (meshes, other frames) appear in their new position or orientation.



The D3DX Animation System

Figure 9.7

So we actually have several timers at work here. The global time of the animation controller is the time that is updated every time our application calls the ID3DXAnimationController::AdvanceTime method.

We pass in an elapsed time value that is used to update the controller's timer variable. This same elapsed time is also added to the timers of each active track on the mixer (a track that has an animation set assigned to it). We can think of these as being global times for a given track. By keeping a global time for each track, we have the ability to advance the time of a single track without affecting any of the others. If we only had a single global time shared by all tracks, altering the position of the global time would cause all tracks to be updated.

Finally, while we might think that the track time is the time used by the animation set to fetch its keyframes, this is not the case. The track time is still a global time value that has no consideration for the duration of the animation or whether it is used to play in a looping or ping-pong fashion. Therefore, once the timer of a given track has been updated, the controller will use the ID3DXAnimationSet::GetPeriodicPosition method to map the track time into a local time for the animation. This local time is referred to as the *periodic position* of the animation and it is this periodic position that is passed into the ID3DXAnimationSet::GetSRT method by the controller to fetch the SRT data for a given animation. We will discuss the periodic position of an animation set and the mapping of track time into local animation time more a bit later in the chapter. Just know for now, that this is all handled on our behalf by the animation controller.

10.5 SRT vs. Matrix Keyframes

Earlier we discussed the fact that X file keyframe data can be stored in one of two ways: SRT transforms can be maintained in separate lists or they can be combined into a single list of 4x4 matrices. Regardless of the file keyframe format, the ID3DXKeyframedAnimationSet always stores keyframe data in SRT format. Internally, the D3DXKeyframedAnimationSet maintains a list of private arrays for **each** animation, similar to what we see below:

```
LPCSTR                  AnimationName;
LPD3DXKEY_VECTOR3       pTranslationKeys;
LPD3DXKEY_QUATERNION    pRotationKeys;
LPD3DXKEY_VECTOR3       pScaleKeys;
ULONG                   NumTranslationKeys;
ULONG                   NumRotationKeys;
ULONG                   NumScaleKeys;
```

Each animation maintains three separate lists of keyframe data and stores an animation name so that it knows which frame in the hierarchy these keyframe lists affect. The name of the animation will be the name of the frame to which it is attached. Translation and scaling keyframes are both stored in D3DXKEY_VECTOR3 arrays. This structure contains a time value and a 3D vector:

```
typedef struct _D3DKEY_VECTOR3
{
      FLOAT       Time;
      D3DXVECTOR3 Value;
} D3DKEY_VECTOR3, *LPD3DXKEY_VECTOR3;
```

When used to store translation keyframes, the 3D vector contains the position of the frame relative to its parent frame. When this structure is used to store scale keyframes, the 3D vector stores scaling amounts for the frame matrix along the X, Y, and Z axes.

Rotation keyframes are stored as an array of D3DXKEY_QUATERNION structures. This structure includes a timestamp in addition to a standard quaternion.

```
typedef struct _D3DKEY_QUATERNION
{
      FLOAT             Time;
      D3DXQUATERNION    Value;

} D3DXKEY_QUATERNION, *LPD3DXKEY_QUATERNION;
```

The D3DXQUATERNION structure is a 4 float structure (X,Y,Z,W) that will store the frame orientation relative to the parent. D3DXLoadMeshHierarchyFromX takes care of converting quaternions stored in the X file in WXYZ format to XYZW format before storing them in the interpolator quaternion array.

> **Note:** 3D Studio Max™, Milkshape™ and other animation editors also store their keyframe data in SRT format, not as a single array of matrices. If you are familiar with 3D Studio Max™ you will notice that on the timeline bar, when a keyframe is registered, there are three different colored blocks which denote the type of 'event' is tracked at that time (scale, rotation, or translation). If only a translation occurred, there will only be one colored block there. The animation information layout in the .3DS file format also stores its keyframe animation in an almost identical manner to that discussed above.

It is worth noting that even if your original keyframe data was stored as a matrix list, those matrices would be split into SRT components before they are stored in the animation's SRT arrays by D3DX. Why do this? After all, matrix keyframes seem like a nice idea since we only have to maintain a single array of keyframe data.

As it turns out, there are actually a few good reasons why SRT lists are preferred over matrices. One of the more obvious is memory footprint. There are 16 floats in a matrix keyframe and a maximum of 10 in an SRT keyframe (and perhaps fewer depending on which components are active).

But more importantly, accurate interpolation between matrices is essentially not possible under certain circumstances. This is a direct result of the fact that scaling data and rotation data become intermixed, and are ultimately indistinguishable in the upper 3x3 portion of the matrix.

Consider the following example animation data. In our editing package, we create two cubes side by side. One of these we wish to **scale** by a factor of (`X:-1, Y:1, Z:-1`) over the course of the animation ('the animation' being a simple two keyframe animation) and the other is **rotated** 180 degrees around the Y axis. So in SRT keyframe format, we would end up with two keyframes for each cube:

```
Animation {

  { Cube01 }

  AnimationKey {
```

```
    1;      // 1 = Scale
    2;      // 2 Key frames
    0;     3;  1.000000, 1.000000,  1.000000;;,
    67200; 3; -1.000000, 1.000000, -1.000000;;;
  }
}

Animation {

  { Cube02 }

  AnimationKey {
    0;      // 0 = Rotation
    2;      // 2 Key frames
    0;     4; 0.000000, 0.000000, 0.000000, 1.000000;;.
    67200; 4; 0.000000, 1.000000, 0.000000, 0.000000;;;
  }
}
```

This works fine and we can see that when we run the animation, each cube acts as we would expect it to when the animation is played out. Interpolation of each cube's animation data will produce expected results for any time within the timeline. In the case of the first cube, two scale keyframes are being interpolated, causing the cube to slowly invert itself over the duration of the timeline. In the case of the second cube, two rotation keys will be used for interpolation, causing the cube to turn 180 degrees over the duration of the timeline. If we were to switch over to matrix keys however, let us take a look at the matrix keyframe values that would now be stored in the X file:

```
Animation {
  { Cube01 }
  AnimationKey {
    4;      // 4 = Matrix Key
    2;      // 2 Key frames
    0;     16; {MatrixA};;,
    67200; 16; {MatrixB};;;
  }
}

Animation {
  { Cube02 }
  AnimationKey {
    4;      // 4 = Matrix Key
    2;      // 2 Key frames
    0;     16; {MatrixA};;,
    67200; 16; {MatrixC};;;
  }
}
```

The matrices listed above are just placeholders, so let us have a look at the actual values. MatrixA and MatrixB describe the two matrix keyframes of the first frame (which is to be scaled) and MatrixA and MatrixC describe the two matrix keyframes for the second animated frame (which is to be rotated).

```
{MatrixA}    1.000000, 0.000000, 0.000000, 0.000000,
             0.000000, 1.000000, 0.000000, 0.000000,
             0.000000, 0.000000, 1.000000, 0.000000,
             0.000000, 0.000000, 0.000000, 1.000000

{MatrixB}   -1.000000, 0.000000,  0.000000, 0.000000,
             0.000000, 1.000000,  0.000000, 0.000000,
             0.000000, 0.000000, -1.000000, 0.000000,
             0.000000, 0.000000,  0.000000, 1.000000

{MatrixC}   -1.000000, 0.000000,  0.000000, 0.000000,
             0.000000, 1.000000,  0.000000, 0.000000,
             0.000000, 0.000000, -1.000000, 0.000000,
             0.000000, 0.000000,  0.000000, 1.000000
```

We can see that the first cube (the cube we are attempting to scale by -1 on both the X and Z axes) uses both MatrixA and MatrixB as its keyframes. MatrixA is an identity matrix (for both frames) so the cube meshes attached to those frames will start off aligned with the world X, Y, and Z axes facing down +Z. We can see that MatrixB is a scaling matrix. At 67200 ticks, the first cube will scale itself such that it will then be inverted and facing down -Z. So far, this is all essentially fine.

However, have a close look at MatrixC which is used for the second cube's final keyframe. This defines our 180 degree Y rotation keyframe. You will notice that MatrixB and MatrixC contain the exact same values. In fact, a two axis negative scaling matrix is identical to a 180 degree rotation matrix on the third (unused) axis. At first you might think that this is not a problem. After all, the final result you want in both cases is identical, and thus the matrices should be identical too, right? Certainly, in both cases, at timestamp 67200, both cubes would be facing down the -Z axis and would look identical.

The problem occurs when the animation set for which the animation is defined is trying to interpolate between the two keyframes (A/ B and A/C) to generate the frame matrix for a time that falls between their timestamps. How would the interpolation process know whether it should gradually scale the object from one keyframe to the next or whether it should rotate it between keyframes? Take a look at the following two matrices which show how the matrices in between the keyframes in our example might look, depending on whether the two keyframes are supposed to represent a scale or a rotation:

The first matrix is 50% through a 180 degree rotation about the Y axis. This is the frame matrix we would expect the interpolation process to generate for our rotating cube frame exactly halfway through its timeline:

```
0.000000, 0.000000, -1.000000, 0.000000,
0.000000, 1.000000,  0.000000, 0.000000,
1.000000, 0.000000,  0.000000, 0.000000,
0.000000, 0.000000,  0.000000, 1.000000
```

The second matrix is 50% through a -1.0 scaling along the X and Z axes. This is the frame matrix we would expect the interpolation process to generate for our scaling cube frame exactly halfway through its timeline.

```
0.000000, 0.000000, 0.000000, 0.000000,
0.000000, 1.000000, 0.000000, 0.000000,
0.000000, 0.000000, 0.000000, 0.000000,
0.000000, 0.000000, 0.000000, 1.000000
```

These two matrices are clearly quite different, even though carrying on interpolating all the way to the end of the animation (100%) would result in identical matrices, as we have seen. This shows that while the keyframe matrices of both a 180 degree Y axis rotation and a –1.0 XZ axis scale are identical, the matrices generated by the interpolation at times that fall between the two keyframes can be very different.

So how exactly does D3DX address this problem when converting matrix keyframes stored in an X file to separate SRT lists for the animation stored inside the animation set? How does it look at two keyframe matrices (such as those in our example) and determine whether these two keyframes represent a scale or a rotation? The answer is: it does not know. There is no way to distinguish between our scale and rotation examples because in both cases the two keyframes for each animation are identical, even though we would desire different intermediate matrices during animation.

At load time, D3DX will extract what rotation, translation, and scale data it can from the matrix keyframes and store them in SRT format and hope they are correct. It will simply assume, in a case like we see above, that rotation is the correct animation and store the data in the animation's rotation keyframe list. This is generally fairly safe to assume as rotation animations are typically more common than scaling animations.

This matrix keyframe ambiguity is precisely why most modeling applications export keyframes in SRT format. Matrix keys are not completely useless however. Certain animations that require transforms that cannot easily be represented with the SRT approach (skew matrices, projection matrices, etc.) can benefit from a matrix keyframe list. In fact, this is why keyframe matrices were added to DirectX 8.0 (they did not exist in DirectX prior to that point).

There is a way around some of these matrix keyframe problems. If we were to use three key-frames for these cases instead of two, inserting our '50%' matrices outlined above into the animation, we would end up with a more accurate representation of the desired animation type. It would still have significant issues, but the more keys we add, the more accurate the animation will become. There are other cases, other than the one outlined above, where the use of matrix keys can cause problems, but generally, by sampling more keyframes during the export of the animation data, these problems can be reduced.

The bottom line is, it is probably best to favor SRT format if you have a choice when exporting your X files. Only choose a matrix format if you are sure that they will not present a problem.

10.6 The D3DX Animation Interfaces

In this next section we will examine the D3DX animation interfaces that our application will need to work with in order to animate frame hierarchies. Rather than starting at the top level of the system and covering the ID3DXAnimationController interface first (which is often the only interface our application will need to work with extensively), we will take the reverse approach and study the ID3DXKeyframedAnimationSet first. This will give us a better idea about how the animation controller uses the interface of its registered animation sets to generate the new matrices for our frame hierarchy. This discussion is not superfluous. If we understand how the system works internally, we will be better informed to use it wisely.

When we load an X file that contains keyframe animation data, one or more of these animation sets will be created for us by D3DXLoadMeshHierarchyFromX and will be registered with the animation controller. Each animation set contains a collection of one or more animations. Each animation is really just a named set of keyframes that apply to a frame in the hierarchy with a matching name when the animation set is played.

Our application will rarely need to work with the methods of the ID3DXKeyframedAnimationSet interface directly. Often, our only exposure comes when using the interface to bind animation to a track on the animation controller's internal mixer. For example, we will frequently use the ID3DXAnimationContoller interface to return one of its registered animation sets, by name or index. We will get back an ID3DXAnimationSet interface (the base class interface) which we can pass into ID3DXAnimationController::SetTrackAnimationSet to bind that animation set to a track on the mixer. Regardless of how many animation sets are registered with the animation controller, only animation sets assigned to active tracks on the animation mixer will influence the hierarchy on the next call to ID3DXAnimationController::AdvanceTime.

10.6.1 The ID3DXKeyframedAnimationSet Interface

In DirectX terminology, an *animation* is a set of keyframe lists for a single frame in the hierarchy. The *animation set* stores and manages a set of animations. Animation sets are in turn managed by the *animation controller*, which itself is the top level interface in the D3DX animation system. You may recall that this was the case based on our earlier discussions about the way the data was stored inside the X file. In our X file we saw that an AnimationSet data object contained a number of Animation data objects. These Animation data objects are the file-bound cousins of an animation in an animation set, containing keyframe data for a given frame in the hierarchy.

> **Note:** Many currently available X file exporters for 3D Studio Max™ and Milkshape™ save their data into a single animation set. This means that, without modification, these files will cause D3DXLoadMeshHierarchyFromX to create an animation controller with a single animation set containing all of the keyframe interpolators in the file. In Lab Project 10.2 we will develop a tool that allows us to modify X files so that they can contain multiple animation sets.

The ID3DXKeyframedAnimationSet mirrors the AnimationSet data object in the X file -- storing one or more animations. When D3DX loads an X file that contains animation data, it will automatically create all of the animations and incorporate them into their appropriate animation sets.

ID3DXAnimationSet exposes many methods, most of which we are not likely to need very often. This is because, when playing back animations, the animation controller will interact with the animation sets (and their underlying animations) on our behalf. All we have to do is call the controller's AdvanceTime function and all of this happens automatically. However, the interface also exposes methods to retrieve the underlying keyframe data for each of its animations so that one can directly examine the keyframes assigned to a given frame in the hierarchy. While this is not something you are likely to do on a frequent basis, it might come in handy during debugging. Studying this interface will also teach us how the entire D3DX animation system works at a low level and will provide us the knowledge to build and populate our own animation sets programmatically if we wish.

The collection of animations stored in an animation set defines the purpose of the set. Sometimes a single animation set will animate every frame in the hierarchy and under other circumstances only select frames may be affected. The latter is useful when we only wish to update portions of an articulated structure. A character is a good example; we could separate the character into logical areas (legs, torso, arms, head, etc.) and apply particular animation sets to individual parts. Consider the following animation set types for an example game avatar:

Animation Set: 1
Name: Walk
Description: This animation set contains animations which animate the leg frames of the character hierarchy to emulate the act of walking. No other animations are included for any other portion of the character.

Animation Set: 2
Name: Run
Description: Identical to the Walk animation set in that it only contains animations which animate the legs of the character. In this case, the legs are animated to give the impression that the character is running.

AnimationSet: 3
Name: Attack
Description: This animation set contains animations which animate the characters arms and torso to swing his sword.

While many more possible animation sets can exist, and some of these may even be too restrictive, the key point here is the isolation of important areas in the hierarchy. The more we can do this, the more effective will be our use of the animation mixer. As we will see later, we will be able to mix multiple animation sets together on the fly, which means that we can have our character run and attack simultaneously, without the artist having to create a specific 'Run+Attack' animation set.

You are reminded that each animation owned by an animation set may have its own unique animation length and that each animation runs concurrently. If you would prefer that all animations in a given set

finish execution simultaneously, you can do this quickly in your editing package by adding a final keyframe (at the same point in the timeline) for all frames that are animated. Remember that the duration of the animation set is the length of its longest animation (i.e., it is determined by the keyframe with the largest time stamp). If the animation set is configured to play continuously, then the animation will only loop (or ping-pong) around once this keyframe has been executed. If other animations in the set have reached their final keyframe some time before, they will remain inert until the animation set timeline restarts. The length of an animation set is referred to by DirectX as its *period* and the current position within the local timeline of an animation set is referred to as its *periodic position*.

This ID3DXKeyframedAnimationSet interface is derived from the ID3DXAnimationSet interface, which is essentially an abstract base class derived from `IUnknown`. ID3DXAnimationSet does not implement any functionality itself, but serves as the base for `ID3DXKeyframedAnimationSet` and `ID3DXCompressedAnimationSet`. It can also be inherited in order to implement your own custom SRT interpolation system if desired. Generically speaking, an animation set must generate the scale, rotation, and translation keyframe values for one of its animations based on a specified time value passed into its GetSRT member function. It returns that information via the base class functions to the animation controller.

The animation controller is only concerned with the methods exposed in the base interface, and as such, any interfaces must implement all of these methods. The ID3DXKeyframedAnimationSet interface would be useless to the controller if it did not implement the GetSRT method of the base interface. However, the ID3DXKeyframedAnimationSet interface exposes many other functions that our application can use to examine or set the keyframe data. When D3DXLoadMeshHierarchyFromX loads in animation data of the type discussed above, it will automatically create keyframed animation sets for us and register them with the returned controller. The exception to the rule is if the animation is stored in the X file in compressed format. In this case, a compressed animation set will be created instead. The controller works with both interfaces in exactly the same way, since they both expose the base class functionality that the controller desires. It is currently very rare to find X files that contain compressed animation data, so we will usually be dealing with the ID3DXKeyframedAnimationSet interface.

The base animation set interface, ID3DXAnimationSet, exposes eight methods. Each is implemented by the ID3DXKeyframedAnimationSet. While the animation controller will work with these on our behalf most of the time, if you intend to plug your own animation set interpolation system into the D3DX animation system, your derived class will need to implement these functions. This means knowing what the animation controller will expect in return when it calls them.

Base Methods

```
LPCSTR GetName(VOID);
```

This function takes no parameters and it returns a string containing the name of the AnimationSet. D3DXLoadMeshHierarchyFromX will create D3DXAnimationSet object(s) and properly assign them names based on the matching AnimationSet data object name in the X file.

The following example is the beginning of an AnimationSet data object in an X file which has been assigned the name ‘Climb’. The D3DXAnimationSet for this object will be assigned the same name, and calling ID3DXAnimationSet::GetName would thus return the string ‘Climb’.

```
AnimationSet Climb
{
      child ‘Animation’ objects defined here
}
```

You may recall from our earlier examination of X files that we are not required to assign names to data objects. Therefore it is possible that D3DX will load animation sets that have not been explicitly named in the X file. In this case, the corresponding D3DXAnimationSet will have an empty name string. If you have the choice (in your editing package or X file exporter) to assign animation sets meaningful names, it is helpful to do so. It allows you to reference the different animation sets through the animation controller using the actual name of the animation set (such as ‘Walk’ or ‘Climb’) instead of relying on a numeric index.

`DOUBLE GetPeriod (VOID);`

This function returns the total amount of time (in seconds) of the longest running animation in the animation set. The duration of an animation set is referred to as its period. If animation 1 runs for 14 seconds, but animation 2 runs for 18 seconds, this function will return 18. Indeed the first animation will have finished 4 seconds previously during playback. Only when the period value is reached in an animation set’s local timeline, will its timeline be reset back to zero and both animations would being animating again (assuming the animation set is configured to loop or ping-pong). The period of the animation set is equal to the scale, rotate, or translation key with the highest timestamp among all animations in the set. If we have a scale key, a rotation key and a translation key with timestamps of 50 ticks, 150 ticks and 20 ticks, the period of the animation set would be 150/TicksPerSecond.

`DOUBLE GetPeriodicPosition ( DOUBLE Position );`

This function is used by the animation controller to get the current position within an animation set’s local timeline, given a global track time (as discussed previously). Every time we pass an elapsed time value into the ID3DXAnimationController::AdvanceTime function, the animation controller adds this input time to the timer of each mixer track. Thus, the mixer track timers continue to increment with each call to AdvanceTime. When the animation controller needs to fetch the SRT data for an animation from a given animation set, it will use this method to map the track time of the animation controller into the animation set’s local timeline. This is because the track time is always incremented when we call AdvanceTime and therefore it may be well outside the period of the animation set.

If the animation set is set to loop or ping-pong, then the animation set must not simply stop when the track time exceeds its period. Rather, the track time should be mapped into a local time (a periodic position) that is within the range of its keyframe data. It is this time that should be returned to the controller so that it can be used to fetch SRT data. Therefore, studying the above function, we can see that the Position parameter would contain a track time that would be passed in by the controller. This

track time needs mapping into an animation set's local time. Again, this local time is referred to as the periodic position.

This function essentially provides a mechanism for the animation set to use the track position to map into its own time (or not) however it sees fit. As the controller has no idea what data is contained in the animation set or whether or not it is set to loop, this function allows the animation set to control that process without involving the controller in the specifics. All the controller cares about is getting back a time value that it can use to fetch SRT values. How this time value is generated from the track position is entirely up to the animation set. For a non-looping animation, this function might simply test to see if the track time is larger than its highest keyframe timestamp. If so, it can pass back the timestamp of the last keyframe in the animation. This would mean that once the track time had exceeded the period of the animation set, it would appear to stop or freeze in its final position, because in every call to AdvanceTime thereafter, the period of the animation set would be returned. This would always generate the same SRT data, with the animation set in its final position.

How the return value of this function is generated depends on whether the animation set is configured to loop or ping pong. For example, imagine that we have called AdvanceTime many times and that the animation controller's track time (the timer of the track the animation set is assigned too) is now set to 155 seconds. Also imagine that the animation set which we are using is configured to loop each time its end is reached. Also, let us assume that this animation has a period of 50 seconds. The animation controller will pass this function the track time (155) and will get returned a value of 5 seconds. This is because at 155 global seconds, the animation set has looped three times and is now on the 5th second of its 4th iteration. Of course, the value returned would be different if the animation set was configured to ping-pong. Ping-Ponging is like looping except, when the animation set reaches the end, the timeline is traversed backwards to the beginning. When the beginning is reached, it starts moving forward again, and so on. Therefore, for every even loop the timeline is moving forward and for every odd loop it is moving backwards (assuming 0 based loop indexing). If we were to call this method on a ping-pong animation, we would get back a periodic position of 45 seconds. This is because it is beginning its fourth cycle and would be moving backwards (5 seconds from its total period of 50 seconds).

The following code might be used by the animation controller when it needed to fetch the SRT data for the first animation in an animation set during a call to AdvanceTime. TrackTime is the current time of the track (accumulated during AdvanceTime calls) which, in our example, is currently at 155 seconds. Do not worry about how the GetSRT function works for the time being; we will cover it in a moment. Just know that it fills the passed scale and translation vectors and the rotation quaternion with the correct interpolated keyframe data given the input local animation set time.

```
D3DXVECTOR3      Scale , Translate; // Used to store animations scale and position
D3DXQUATERNION   Rotate;            // Used to store animations new orientation

AnimationIndex    = 0;   // Get SRT data for first animation in set
TrackTime         = 155; // Accumulated global time of the track.

AnimSetLocalTime  = pAnimSet->GetPeriodicPosition ( TrackTime )
pAnimSet->GetSRT( AnimSetLocalTime, AnimationIndex, &Scale, &Rotate, &Translate );
```

The above code should give you a pretty good idea of the interaction between the animation controller and its animation sets. The important point here is that the animation set's GetSRT function expects to

be given a local time value which is fully contained within the range of time specified by its keyframe data. GetPeriodicPosition performs this mapping from a global time value, giving the animation controller the correct local time value to pass into the GetSRT function.

`UINT GetNumAnimations(VOID);`

This function accepts no parameters and returns an unsigned integer containing the number of animations being managed by this animation set. This tells the animation controller (or any caller for that matter) how many frames in the hierarchy are being manipulated by this animation set. Each animation in the animation set contains a frame name and SRT keyframe lists for that frame.

This function will be called by the animation controller during the call to AdvanceTime. The controller can then loop through these animations and call ID3DXAnimationSet::GetSRT for each one. This will generate the new SRT data for each frame in the hierarchy animated by this animation set. This SRT data is then used to build the matrix for each of these frames.

`HRESULT GetAnimationIndexByName( LPCSTR pName, UINT*pIndex )`

There may be times when your application wants to fetch the keyframe data for an individual animation within an animation set. You will see in a moment that there are methods of the ID3DXKeyframedAnimationSet interface that allow us to do this, but they must be passed the index of the animation. Often, your application might not know the index but will know the name of the animation/frame. Using this function we can pass in the name of the animation we wish to retrieve an index for and also pass the address of an unsigned integer. On function return, if an animation exists that animates the frame name, the index of this animation will be returned in the pIndex parameter. It is the index of the animation that the animation controller passes to the ID3DXAnimationSet::GetSRT function.

`HRESULT GetAnimationNameByIndex( UINT Index, LPCSTR *ppName );`

This function is the converse for the above one. If you know the index of an animation within an animation set, you can use this function to retrieve its name. As the first parameter you pass in the index of the animation whose name you wish to know, and in the second parameter you pass in the address of a string pointer. On function return, the string will contain the name of the animation. As this is also the name of the frame in the hierarchy to which the animation is attached, we can imagine how this could be very useful.

The animation controller itself might well use this function inside its AdvanceTime method. As mentioned previously, the AdvanceTime method will loop through each animation stored in the animation set and call GetSRT for it. Once this SRT data has been combined into a new matrix, that matrix should replace the parent-relative matrix data currently existing in the assigned frame. At this point, the animation controller has the new matrix and the index of the animation it was generated for, but does not yet have its name.

The following code snippet shows how the animation controller might use this function to get the name of the animation which it then uses to find the matching name in the hierarchy. Much of the detail is left out (such as the call to GetSRT for each animation and the construction of the new matrix) but hopefully you will get the overall idea.

```
for ( UINT i = 0 ; i < pAnimSet->GetNumAnimations(); i++ )
{
     …
     …
     // GetSRT would be called here and the new matrix (mtxFrame )
     // for this animation would be  generated
     …
     …
     LPCSTR pName = NULL;
     pAnimSet->GetAnimationNameByIndex( i, &pName );
     pFrame = D3DXFrameFind(m_pFrameRoot, pName );

     // Store the new matrix in the frame hierarchy
     pFrame->TransformationMatrix = mtxFrame;
}
```

The next function, GetSRT, is really the core of the interpolation system.

```
HRESULT GetSRT
(
    DOUBLE           PeriodicPosition,
    UINT              Animation,
    D3DXVECTOR3      *pScale,
    D3DXQUATERNION   *pRotate,
    D3DXVECTOR3      *pTranslate
);
```

This method is usually called by the animation controller to fetch the SRT data for a specified frame in the hierarchy. If no other animation set is being used which has an animation assigned to the same frame in the hierarchy, then this SRT data will be placed directly into the hierarchy frame matrix. Therefore we might say that this method generates the new matrix data for a specified frame in the hierarchy for a specified local time. If multiple animation sets are assigned to the animation mixer which animate the same frame in the hierarchy, then the resulting SRT data for that frame from **each** animation set will be blended together (by the animation controller) based on the current mixer track settings. The resulting blended data will then be placed into the frame matrix.

This function is passed the time (in seconds) in the local timeline of the animation that we would like to have the SRT data returned for. In other words, this function expects to be passed the periodic position in the animation set for which SRT data will be generated for the specified animation. We also pass in an integer as the second parameter describing the animation we would like the SRT data returned for. Remember, an animation is just a named container of keyframe data that is assigned to a frame in the hierarchy. Therefore, if the animation controller (or our application) would like to get the SRT data for a frame in the hierarchy called ‘Wheel_Frame’ for a global time of 406 seconds, we could do something like we see in the following code:

```
UINT              Animation ;
D3DXVECTOR3       Scale , Translate;
D3DXQUATERNION    Rotate;

// Get the index of the frames animation in the animation set
pAnimSet->GetAnimationIndexByName("Wheel_Frame" , &Animation );

// Map ever increasing track time to local time taking into account
// looping or ping-ponging
PeriodicPosition = pAnimSet->GetPeriodicPosition( 405 );

// Get the SRT data for that animation at 405 global seconds
pAnimSet->GetSRT( PeriodicPosition , Animation , &Scale , &Rotate , &Translate);

// Here the animation controller would use Scale, Rotate and Translate to
// build a new frame matrix
```

As the above code demonstrates, if you do not know the index of the animation within the animation set then you can use the GetAnimationIndexByName method to retrieve the index of the animation/frame with the specified name. We can then pass this index into the GetSRT Method. The code above is very similar to the call made by the animation controller to an animation set to fetch the SRT data for each of its animations.

Notice how we also pass in the address of a scale vector, a rotation quaternion and a translation vector. On function return, these three variables will be filled with the SRT data that can then be used to rebuild the matrix.

> **Note** : Our application will not need to call the GetSRT method directly to generate the SRT data for each animation. The animation controller will handle these calls on our behalf when we call ID3DXAnimationController::AdvanceTime. However, you do have access to the interface for each animation set, so your application could use an animation set object as a simple keyframe interpolator even if your application was not using the D3DX animation controller. For example, if you have your own animation system in place, you could manually create a D3DXKeyframedAnimationSet object, fill it with keyframe data and use it to interpolate SRT data for your own proprietary system.

In summary, when we call ID3DXAnimationController::AdvanceTime, the animation controller will loop through each of its active animation sets. For each animation set it will loop through each stored animation. For each animation it will call ID3DXAnimationSet::GetSRT. This function will return the scale, rotation and position that the assigned frame in the hierarchy should have based on the current time. When this process is finished and the SRT data for all animations in all active animations sets is handed back to the controller, it will then check to see if multiple SRT data exists for the same frame in the hierarchy. This can happen when two animation sets have an animation that animate the same frame in the hierarchy. When this is the case, the multiple SRT data sets for that frame are blended together and used to construct the final frame matrix. When only one set of SRT data exists (because a hierarchy frame is being animated by only a single animation set) the SRT data is used to build the frame matrix directly. However, the SRT data will still have the properties of the mixer track to which it is assigned applied to it. Therefore, if the mixer track to which the animation set was assigned was configured to scale the SRT data by 0.5 for example, that scaling process will still occur regardless of whether multiple animations are used or not. We will cover the animation mixer later when we examine the ID3DXAnimationController interface.

So we now know how the ID3DXAnimationSet::GetSRT function is used, but how does it return the SRT data for the requested animation? Although we do not have access to the original DirectX source code, we can, for the purpose of understanding the methodology, write our own version of the function. We will do this at the end of this section. This will demonstrate not only how the GetSRT function works (in case you need to derive your own animation set interfaces for use with the D3DX animation system), but also how it fits into grand scheme of things.

Keyframe Retrieval Functions

The ID3DXKeyframedAnimationSet interface also has several functions that allow an external process to query its underlying keyframe arrays. We can query the number of scale, rotate and translation keys that might exist for a given animation in the set or even retrieve those keyframes for examination. Usually, our application will not need to do this, but these functions can be useful for debugging purposes or for applications that need to manipulate keyframe data directly (like the animation set splitter application we will create in lab project 10.2).

```
UINT GetNumScaleKeys( UINT Animation );
```

This function takes a single parameter, which is the index of the animation we are querying. It returns an unsigned integer describing the number of scale keyframes stored in the animation set for the specified animation. We can use this function to determine how big an array of D3DXKEY_VECTOR3s we need to allocate if we need to retrieve the data (via a call to GetScaleKeys which will be covered in a moment).

```
UINT GetNumRotationKeys( UINT Animation );
```

This function takes a single parameter, which is the index of the animation we are querying. It returns an unsigned integer describing the number of rotation keyframes stored in the animation set for the specified animation. We can use this function to determine how big an array of D3DXKEY_QUATERNIONs to allocate if we need to retrieve the data (via a call to GetRotationKeys which will be covered in a moment).

```
UINT GetNumTranslationKeys( UINT Animation );
```

This function takes a single parameter, which is the index of the animation we are querying. It returns an unsigned integer describing the number of translation keyframes stored in the animation set for the specified animation. We can use this function to determine how big an array of D3DXKEY_VECTOR3s we need to allocate if we need to retrieve the data (via a call to GetTranslationKeys which will be covered in a moment).

HRESULT GetScaleKeys(UINT Animation, LPD3DXKEY_VECTOR3 pKeys);

This function is called to retrieve all scale keyframes stored within the animation set for the specified animation. We must pass the index of the animation we wish to enquire about and pointer to a pre-allocated array of D3DXKEY_VECTOR3 structures large enough to store every scale keyframe. The scale keys are then copied into this array by the function. The following code demonstrates this concept:

```
D3DXKEY_VECTOR3 *pSKeys = NULL;
pSKeys = new D3DXKEY_VECTOR3 [ pAnimSet->GetNumScaleKeys() ];
pAnimSet->GetScaleKeys( pSKeys );

… //Do something with key data here such as read it or copy it….

delete [] pSKeys;
```

Keep in mind that you are getting back a *copy* of the keyframe data, so modifying the data will not affect the behavior of the interpolator.

HRESULT GetScaleKey(UINT Animation, UINT Key,
LPD3DXKEY_VECTOR3 pScaleKeys);

This function is similar to the previous function except it allows us to retrieve only a single specified scale keyframe. We pass in the index of the animation we wish to retrieve the keyframe for, the index of the keyframe in the animation's keyframe array we wish to retrieve, and the address of a D3DXKEY_VECTOR3 structure. If the animation index and the key index parameters are valid, on function return the input structure in the third parameter will contain the scale keyframe at the specified index (UINT Key) in the specified animation (UINT Animation).

HRESULT GetRotationKeys (UINT Animation, LPD3DXKEY_QUATERNION pKeys);

This function is called to retrieve all rotation keyframes stored within the specified animation within the animation set. We must pass in the index of the animation we wish to retrieve the rotation keyframes for, and a pointer to a pre-allocated array of D3DXKEY_ QUATERNION structures large enough to store every rotation keyframe. The rotation keys are then copied into this array by the function.

HRESULT GetRotationKey(UINT Animation, UINT Key,
D3DXKEY_QUATERNION pRotationKeys) ;

This function is the single key version of the previous function. We pass in the animation index, the index of the rotation keyframe we wish to retrieve, and the address of a D3DXKEY_QUATERNION to fill. Remember that this is a copy of the keyframe you are being returned, so modifying it will not alter the behavior of the animation.

```
HRESULT GetTranslationKeys ( UINT Animation, LPD3DXKEY_VECTOR3 pKeys );
```

This function is called to retrieve all translation keyframes stored within the animation set for the specified animation. We pass the index of the animation we wish to retrieve the position keyframes for and a pointer to a pre-allocated array of D3DXKEY_VECTOR3 structures large enough to store every translation keyframe. The translation keys are then copied into this array by the function

```
HRESULT GetTranslationKey( UINT Animation, UINT Key,
                           LPD3DXKEY_VECTOR3 pTranslationKey );
```

This is the single key version of the function previously described. We pass in the index of the animation we wish to retrieve the keyframe for, the index of the translation keyframe we wish to retrieve, and the address of a D3DXKEY_VECTOR3. On successful function return, the vector will contain a copy of the requested key.

Changing Key Properties

The ID3DXKeyframedAnimationSet interface also exposes three methods (one for each keyframe type) that allow the changing of keyframe values on the fly. Each function takes three parameters. The first parameter is the index of the animation you wish to alter the key value for. The second parameter is the index of the key in the animation's S, R, or T lists that you wish to modify. The final parameter is the new key itself. This final parameter should be the address of a D3DXKEY_VECTOR3 structure if you are modifying a scale or translation key, or the address of a D3DXKEY_QUATERNION structure if you are updating a rotation key. These methods are listed next:

```
HRESULT SetRotationKey( UINT Animation, UINT Key,
                        LPD3DXKEY_QUATERNION pRotationKeys );

HRESULT SetScaleKey( UINT Animation, UINT Key,
                     LPD3DXKEY_VECTOR3 pScaleKeys );

HRESULT SetTranslationKey( UINT Animation, UINT Key,
                           LPD3DXKEY_VECTOR3 pTranslationKey );
```

The animation and the key must already exist in the animation set. That is to say, you can not add keys to an animation with these functions; you can only change the values of existing keys.

Removing Keys

Just as we have the ability to change the property of an already defined keyframe, methods are also available that allow for the removal of keyframes from the animation set. Three methods are exposed through which we can unregister scale keys, rotation keys and translation keys respectively. Each of these methods takes two parameters. The first is the index of the animation we wish to unregister a key for, and the second is the index of the key we wish to remove. These methods are shown below.

```
HRESULT UnregisterRotationKey( UINT Animation, UINT Key );

HRESULT UnregisterScaleKey( UINT Animation, UINT Key );

HRESULT UnregisterTranslationKey( UINT Animation, UINT Key );
```

It should be noted that you should not be unregistering keys during the main game loop or in any time critical section of your code. These functions are very slow because they require the animation set to resize the S, R, and T arrays (depending on which key you are unregistering). For example, if you use the UnregisterRotationKey function and specify a key index somewhere in the middle of the array, the array will need to be resized. This involves dynamically allocating a smaller array and copying the data from the old array into the new array (minus the key you just unregistered) before releasing the old key array from memory. Typically you might do something like this just after you have loaded an animated X file and wish to remove certain keyframes that are not needed.

Removing Animations

The ID3DXKeyframedAnimationSet interface also exposes a method that allows for the removal of an entire animation and all of its SRT data from the animation set. This might be useful if you loaded an animated X file that animated a frame in your hierarchy that you did not want animated.

```
HRESULT UnregisterAnimation( UINT Index );
```

This function takes a single parameter; the index of the animation you wish to unregister.

As an example, let us imagine that we loaded an animated X file and found that when playing its animation, one of the frames in the hierarchy that it animated was called ‘MyFrame1’. Let us also assume that for some reason we do not wish this frame to be animated by the animation set. After loading the X file, we could use the following code to unregister the animation that animates this frame. The code assumes pAnimSet is a pointer to a valid ID3DXKeyframedAnimationSet interface that has been previously fetched from the animation controller. We will cover how to retrieve interfaces for animation sets when we cover the ID3DXAnimationController method shortly.

```
UINT AnimationIndex;
pAnimSet->GetAnimationIndexByName(“MyFrame1”, &AnimationIndex );
pAnimSet->UnregisterAnimation ( AnimationIndex );
```

Retrieving the Ticks per Second Ratio

As previously discussed, keyframe timestamps in an animation set are defined internally using arbitrary units known as ticks. How many of these ticks are considered to equal one second in real time is largely dependant on the application which exported the animation data and the resolution the artist used on the timeline. A function is exposed by the animation set which allows the application to query the TicksPerSecond ratio in which its keyframe timestamps are defined.

DOUBLE GetSourceTicksPerSecond(VOID);

While our application will rarely need to call this method, the animation set object absolutely must know this value. After all, when the animation controller requires the SRT data for a given frame in the hierarchy, it calls the animation set's GetSRT method. This method accepts a time in seconds (not ticks), so the GetSRT function must covert the passed time in seconds into ticks to find the two bounding keyframes to interpolate between.

10.6.2 A Closer Look at GetSRT

In this section we will discuss how we could write our own GetSRT function for an animation set. This will be helpful if you need to derive your own version of an animation set that you will plug into the D3DX animation system. It will also allow us to better understand what the animation controller and its underlying animation sets are doing when we call ID3DXAnimationController::AdvanceTime.

While this source code is not the exact implementation of ID3DXKeyframedAnimationSet::GetSRT, it will return the same data. In fact, we have tested this code as a drop-in replacement and it works very well. This code is for teaching purposes only. Our goal is to look at and understand the interpolation process for SRT keyframes. You will not need to write a function like this in order to use the D3DX Animation System as it is already implemented in ID3DXKeyframedAnimationSet::GetSRT. Note as well that the application will never even have to call this function, because the animation controller will call it for each frame that is animated by each currently active animation set.

As our example function will not be a method of the ID3DXAnimationSet interface, we will add an extra parameter on the end. This will be a pointer to an ID3DXKeyframedAnimationSet interface. We will use this so that our function can fetch the keyframe data from the animation set. In the real method, this pointer is not needed and neither are the methods to fetch the keyframe data from the animation set since the keyframes are owned by the animation set and are directly accessible.

Remember when looking at the following code that this function has one job: to return the SRT data for a single frame/animation for a specified periodic position in the animation set's timeline. We will look at the code a section at a time. As with the function we are emulating, this function should be passed the periodic position and animation index we wish to retrieve the SRT data for, and should also be passed addresses for a scale vector, a rotation quaternion and a translation vector. On function return we will fill these with the interpolated scale, rotation and position information for the specified time. This function will demonstrate what happens behind the scenes when the animation controller calls ID3DXKeyframedAnimationSet::GetSRT for one of its animation sets for a given animation.

```
HRESULT GetSRT( double Time, ULONG Index, D3DXVECTOR3 * pScale,
                D3DXQUATERNION * pRotate, D3DXVECTOR3 * pTranslate,
                ID3DXKeyframedAnimationSet * pSet )
{
    ULONG i;
    double fInterpVal;
    LPD3DXKEY_VECTOR3 pKeyVec1, pKeyVec2;
    LPD3DXKEY_QUATERNION pKeyQuat1, pKeyQuat2;
```

```
    D3DXVECTOR3 v1, v2;
    D3DXQUATERNION q1, q2;

    // Validate parameters
    if ( !pScale || !pRotate || !pTranslate ) return D3DERR_INVALIDCALL;

    // Clear them out as D3D does for us :)
    *pScale     = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    *pTranslate = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    D3DXQuaternionIdentity( pRotate );

    ULONG ScaleKeyCount       = pSet->GetNumScaleKeys( Index );
    ULONG RotationKeyCount    = pSet->GetNumRotationKeys( Index );
    ULONG TranslationKeyCount = pSet->GetNumTranslationKeys( Index );
```

The first section simply initialises the passed scale and translation vectors to zero and sets the passed quaternion to identity. We then use the passed animation set interface to fetch the total number of scale, rotation, and translation keyframes. As discussed earlier, these three methods take, as their only parameter, the index of the animation we are interested in and return the count values. In the actual ID3DXKeyframedAnimationSet::GetSRT function there would be no need for this call because the keyframe data is stored in the animation set. But our mock function will need to access them in order to simulate the interpolation step.

In the next section we have to convert the passed periodic position into ticks. You will recall from our earlier discussion how each animation set has its keyframe timestamps specified in ticks. As the periodic position passed into the function is in seconds, we must multiply the periodic position by the TicksPerSecond value stored in the X file. You will also recall that there may be an AnimTicksPerSecond data object in the X file describing the timing granularity for the timestamps exported by the modelling application. If none is specified, then a default value of 4800 ticks per second is used. The animation set internally stores this ticks per second value. We can use another ID3DXKeyFramesAnimationSet interface method called ‘GetSourceTicksPerSecond’ to retrieve the number of ticks per second. By multiplying the periodic position (in seconds) by the ticks per second, we get the periodic position in ticks. This is the timing value used to define the keyframe timestamps.

```
    double TicksPerSecond     = pSet->GetSourceTicksPerSecond( );
    double CurrentTick        = Time * TicksPerSecond
```

Notice how in the above code we store the period position converted into ticks in the CurrentTick local variable.

As our function is not a method of ID3DXKeyframedAnimationSet and therefore does not have direct access to its keyframe arrays, the following code will need to fetch the keyframes from the animation set. We therefore allocate the (possible) three keyframe arrays (scale, rotation and translation) using the count values previously fetched, and then pass these arrays into the GetScaleKeys, GetTranslationKeys and GetRotationKeys to fetch copies of the keyframe arrays for the specified animation. Remember, the index of the animation we wish to fetch was passed in as the ‘index’ parameter.

```
    D3DXKEY_VECTOR3 * ScaleKeys = NULL;
    D3DXKEY_VECTOR3 * TranslateKeys = NULL;
    D3DXKEY_QUATERNION * RotateKeys = NULL;
```

```
    if(ScaleKeyCount )
       ScaleKeys = new D3DXKEY_VECTOR3[ ScaleKeyCount ];

    if(TranslationKeyCount)
      TranslateKeys= new D3DXKEY_VECTOR3[ TranslationKeyCount ];

    if( RotationKeyCount )
      RotateKeys = new D3DXKEY_QUATERNION[ RotationKeyCount ];

    if ( ScaleKeyCount )      pSet->GetScaleKeys( Index, ScaleKeys );
    if ( TranslationKeyCount )pSet->GetTranslationKeys( Index, TranslateKeys );
    if ( RotationKeyCount )   pSet->GetRotationKeys( Index, RotateKeys );
```

At this point, we have the keyframe data for the animation we wish to generate interpolated SRT data for. Now it is time to examine each array in turn and perform the interpolation. We will start by looking at the generation of the interpolated scale data.

In order to perform keyframe interpolation, we must find the two keyframes that bound the current tick time. Therefore, we loop through each scale key searching for the correct pair. If we find two consecutive keyframes where the current tick is larger or equal to the first and smaller or equal to the second, then we have found our bounding keyframes. When this is the case, we store these two keyframes in pKeyVec1 and pKeyVec2 and break from the loop.

```
    // Calculate an interpolated scale by finding the
    // two key-frames which bound this timestamp value.
    pKeyVec1 = pKeyVec2 = NULL;
    for ( i = 0; i < ScaleKeyCount - 1; ++i )
    {
        LPD3DXKEY_VECTOR3 pKey     = &ScaleKeys[i];
        LPD3DXKEY_VECTOR3 pNextKey = &ScaleKeys[i + 1];

        // Do these keys  bound the requested time ?
        if ( CurrentTicks >= pKey->Time && CurrentTick <= pNextKey->Time )
        {
            // Store the two bounding keys
            pKeyVec1 = pKey;
            pKeyVec2 = pNextKey;
            break;

        } // End if found keys
    } // Next Scale Key
```

At this point, we have the two scale keyframes which must be interpolated. Next we copy the 3D scale vectors from each keyframe (stored in D3DXKEY_VECTOR3::Value) into 3D vectors v1 and v2. We have discussed interpolation several times in prior lessons and this interpolation is no different. We calulate the time delta between the current tick and the first keyframe's timestamp, and then we divide that by the time delta between both timestamps. This returns a value in the range of 0.0 to 1.0 which can be used to linearly interpolate between the two 3D vectors. The result of the linear interpolation is stored in 'pScale' so that it is accessible to the caller on function return.

```
    // Make sure we found keys
    if ( pKeyVec1 && pKeyVec2 )
```

```
    {
        // Interpolate the values
        v1 = pKeyVec1->Value;
        v2 = pKeyVec2->Value;
        fInterpVal  = CurrentTick - pKeyVec1->Time;
        fInterpVal /= (pKeyVec2->Time - pKeyVec1->Time);
        D3DXVec3Lerp( pScale, &v1, &v2, (float)fInterpVal );
    } // End if keys were found
```

Notice in the above code that we use the 3D vector interpolation function provided by D3DX. D3DXVec3Lerp performs the interpolation between 3D vectors v1 and v2 using the passed interpolation percentage value. This function simply does the following:

V1 + fInterpVal * (V2 – V1)

As fInterpVal is a value between 0.0 and 1.0, we can see how the resulting vector will be a vector positioned somewhere between these two keyframes. We have now successfully generated the S component of the SRT data that the function needs to return.

Our next task is to interpolate the rotation component of the SRT data. We do this is very much the same way. We first loop through every keyframe in the rotation keyframe array (remember that these are stored as quaternions) and search for the two keyframes the bound the current tick.

```
    // calculate an interpolated rotation by finding the
    // two key-frames which bound this timestamp value.
    pKeyQuat1 = pKeyQuat2 = NULL;
    for ( i = 0; i < RotationKeyCount - 1; ++i )
    {
        LPD3DXKEY_QUATERNION pKey     = &RotateKeys[i];
        LPD3DXKEY_QUATERNION pNextKey = &RotateKeys[i + 1];

        // Do these keys bound the requested time ?
        if ( CurrentTick >= pKey->Time && CurrentTick <= pNextKey->Time )
        {
            // Store the two bounding keys
            pKeyQuat1 = pKey;
            pKeyQuat2 = pNextKey;
            break;
        } // End if found keys
    } // Next Rotation Key
```

Now that we have found the two quaternions that describe the rotation to interpolate, we will interpolate between the two quaternions using the D3DXQuaternionSlerp method. Performing a spherical linear interpolation (SLERP) is one of the main reasons quaternions are used to represent rotations. Rather than linearly interpolate between the rotation angles which would cut a straight line from one to the other, a spherical linear interpolation allows us to interpolate along an arc between the rotations. This provides much smoother and more consistent movement from one rotation to the next. If we imagine that the two rotation vectors describe two points on a sphere, slerping between them will follow the path from one point to the next over the surface of the sphere. A linear interpolation would cut a straight line through the sphere from one point to the other.

```
    // Make sure we found keys
    if ( pKeyQuat1 && pKeyQuat2 )
    {
        // Interpolate the values
        D3DXQuaternionConjugate( &q1, &pKeyQuat1->Value );
        D3DXQuaternionConjugate( &q2, &pKeyQuat2->Value );
        fInterpVal  = CurrentTick - pKeyVec1->Time;
        fInterpVal /= (pKeyVec2->Time - pKeyVec1->Time);
        D3DXQuaternionSlerp( pRotate, &q1, &q2, (float)fInterpVal );
    } // End if keys were found
```

Notice how the above code converts the quaternions for compliance with a left-handed coordinate system by using the D3DXQuaternionConjugate function. This function negates the sign of the *x*,*y*,*z* components of the quaternion (the rotation axis), essentially flipping it to point in the opposite direction. The reason we do not need to negate the *w* component (the rotation angle) also is that the D3DX functions expect the *w* component of the quaternion to represent a counter-clockwise rotation about the axis. This is the opposite direction that rotations are interpreted in a matrix. Therefore, by flipping the axis and not changing the sign of *w*, we essentially end up with the rotation axis flipped so that it now represents the same axis in a left-handed coordinate system with a *w* component giving us a clockwise rotation about this axis. Note as well how the result of the slerp is stored in the quaternion structure passed into the function so that it is accessible the caller on function return.

Finally, we interpolate the translation vector using the two bounding keyframes. This code is almost identical to the code that interpolates the scale vectors, so it is shown in its entirety below.

```
    // Calculate an interpolated translation by finding the
    // two key-frames which bound this timestamp value.
    pKeyVec1 = pKeyVec2 = NULL;
    for ( i = 0; i < TranslationKeyCount - 1; ++i )
    {
        LPD3DXKEY_VECTOR3 pKey     = &TranslateKeys[i];
        LPD3DXKEY_VECTOR3 pNextKey = &TranslateKeys[i + 1];

        // Do these keys bound the requested time ?
        if ( CurrentTick >= pKey->Time && CurrentTick <= pNextKey->Time )
        {
            // Store the two bounding keys
            pKeyVec1 = pKey;
            pKeyVec2 = pNextKey;
            break;
        } // End if found keys
    } // Next Translation Key

    // Make sure we found keys
    if ( pKeyVec1 && pKeyVec2 )
    {
        // Interpolate the values
        v1 = pKeyVec1->Value;
        v2 = pKeyVec2->Value;
        fInterpVal  = CurrentTick - pKeyVec1->Time;
        fInterpVal /= (pKeyVec2->Time - pKeyVec1->Time);
        D3DXVec3Lerp( pTranslate, &v1, &v2, (float)fInterpVal );
    } // End if keys were found
```

Finally, with the SRT data correctly interpolated and stored correctly, this version of the function must delete the key arrays that were temporarily allocated to retrieve the keyframe data from the animation set. Obviously, the ID3DXKeyframedAnimationSet::GetSRT method would not need to allocate or deallocate these arrays as it would have direct access to the keys.

```
    if ( ScaleKeys ) delete []ScaleKeys;
    if ( RotateKeys ) delete []RotateKeys;
    if ( TranslateKeys ) delete []TranslateKeys;

    // Success !!
    return D3D_OK;
}
```

There are still a few methods of the ID3DXKeyframedAnimationSet that we have not yet covered. Primarily, the ones we are most interested in for this chapter are those we will use to register callback functions that are triggered by keyframes. We will look at these methods later when we cover the D3DX animation callback system.

10.6.3 D3DXCreateKeyFramedAnimationSet

There will be times when you will be unable to use D3DXLoadMeshHierarchyFromX to load animation data. For example, you may be creating runtime animation data for the animation controller or perhaps loading a custom animation file. In these cases you will need to know how to manually build D3DXAnimationSet objects so that you can register them with the animation controller yourself. D3DX exposes a global function for this purpose called D3DXCreateKeyFramedAnimationSet. The function is shown below with its parameter list followed by a description of each parameter.

```
HRESULT WINAPI D3DXCreateKeyframedAnimationSet
(
    LPCSTR pName, DOUBLE TicksPerSecond,
    D3DXPLAYBACK_TYPE Playback,
    UINT NumAnimations, UINT NumCallbackKeys,
    CONST LPD3DXKEY_CALLBACK *pCallKeys,
    LPD3DXKEYFRAMEDANIMATIONSET *ppAnimationSet
);
```

Let us discuss the parameters to this function one at a time:

LPCSTR *pName*

This function allows you to assign the animation set a name. While not required it is often more intuitive to work with named animation sets. You will see when we cover the ID3DXAnimationController interface shortly, that one of the things we will often want to do is have it return us a pointer to one of its animation sets. While we can fetch animation sets by index, it is often preferable to use ID3DXAnimationController::GetAnimationSetByName function (covered shortly). It is certainly more intuitive when looking at animation code to see references to animation sets such as ‘Walk’ or ‘Run’ instead of indices like 2 or 4. This parameter is analogous to the name that may be assigned to an AnimationSet data object in an X file.

DOUBLE *TicksPerSecond*

When creating an animation set manually, we will be responsible for filling its SRT keyframe arrays. Therefore, the units of measurement used for keyframe timestamps (ticks) are completely up to us. We must inform the animation set, at creation time, the number of ticks per second that we will be using in our keyframe timestamps. This is needed because the GetSRT method (called by the animation controller) is passed a periodic position in seconds which must be converted to ticks. This is the means by which correct bounding keyframes are found.

D3DXPLAYBACK_TYPE *Playback*

This parameter is used to inform D3DX how we would like the mapping between global time (the time of the track the animation set is assigned to) and the animation set's local time (the periodic position) to be performed. You will recall that before the animation controller calls ID3DXAnimatonSet::GetSRT, it must first map the global track time the animation set is assigned to into the animation set's local time using the ID3DXAnimationSet::GetPeriodicPosition function. It is this function that returns the local animation time that is passed into GetSRT. With this parameter we can choose how this function calculates the periodic position of the animation set and thus how the animation set behaves when the track time exceeds the animation sets total period.

For example, we can configure the GetPeriodicPosition method such that if the track time passed in exceeds the period, the periodic position is wrapped around, causing the animation set to continuously loop as the global time climbs ever higher. Therefore our application can repeatedly call ID3DXAnimationController::AdvanceTime without worrying about whether the accumulated global time of the controller (and each of its tracks) is inside or outside the period of the animation set.

Alternatively, we can choose for the mapping to occur such that if the global time exceeds the period of the animation, the animation set essentially stops playing (remains frozen at its last keyframe). Using this mapping, as soon as the track time exceeds the period of the animation set assigned to that track, the animation set stops playing.

Finally, we can make the GetPeriodicPosition method map the track time to a periodic position using ping-ponging. As discussed previously, when this mapping is used, when the animation set period is exceeded, its periodic position starts to backtrack towards the beginning of the local timeline. When the beginning is reached, the timeline direction is flipped again so that the periodic position starts moving forwards towards the end of the timeline, and so on.

We can specify the three possible behaviors of the animation set we are creating using a member of the D3DXPLAYBACK_TYPE enumeration:

```
typedef enum _D3DXPLAYBACK_TYPE {
    D3DXPLAY_LOOP = 0,
    D3DXPLAY_ONCE = 1,
    D3DXPLAY_PINGPONG = 2,
    D3DXEDT_FORCE_DWORD = 0x7fffffff
} D3DXPLAYBACK_TYPE;
```

To better understand how the three modes cause the GetPeriodicPosition function to map global track time to periodic positions differently, let us examine the case of an animation

set that has a period of 25 seconds. Let us also assume that the track time we wish to map to a periodic position (to feed into the GetSRT function) is currently at 30 seconds.

D3DXPLAY_LOOP
If the animation set is created with this option, then when a global track time of 30 seconds is reached by the animation controller, the GetPeriodicPosition method of the animation set will return a periodic position of 5 seconds. This is because the animation set has reach the end and is now five seconds (30 – 25) into its second loop.

D3DXPLAY_ONCE
If the animation set is created with this option, then after 25 seconds the animation would stop playing. The frame matrix would be essentially frozen in the position specified by its final SRT keyframes. At 30 seconds of global track time, the animation would have stopped playing 5 seconds earlier.

D3DXPLAY_PINGPONG
If the animation set is created with this option, then at 30 seconds global track time, the GetPeriodicPosition method would return a periodic position of 20 seconds. This is because the period of the animation set was reached 5 seconds earlier (at 25 seconds) and has backtracked 5 seconds from the end towards the beginning (25 – [30-25]).

UINT *NumAnimations*
When we create an animation set we must specify the maximum number of frames in the hierarchy it should be capable of containing animations for. As previously discussed, each animation contains a frame name and SRT keyframe arrays for that frame. Although we are allocating room for the maximum number of animations at animation set creation time, the keyframe data will be added to each animation as a separate step once it has been created. We shall see this in a moment.

UINT *NumCallbackKeys*
CONST LPD3DXKEY_CALLBACK **pCallKeys*
These parameters will be covered later in the chapter when we discuss the D3DX animation system's callback mechanism. Just know for now that you have the ability to pass in a series of callback keys. Each key contains a timestamp and a collection of data. When one of these callback keys is encountered in the timeline during an animation update, this data can be passed to an application defined callback function that performs some arbitrary task (e.g. playing a sound effect or disabling the animation track). Callback keys are defined per animation set instead of per animation. While they too are defined using ticks, they are handled quite differently by the controller and animation set. For this reason, we will forget about them for now and revisit them later. For now, let is just assume that we will pass in NULL as these two parameters. This informs D3DX that this animation set will not have any callback events registered with it.

LPD3DXKEYFRAMEDANIMATIONSET **ppAnimationSet*
As the final parameter to this function we must pass in the address of an ID3DXKeyframedAnimationSet interface pointer. On successful function return, it will be a pointer to the interface for the newly created animation set object.

An important thing to note is that in order for animation sets to be used, they must be *registered* with an animation controller. This is because the ID3DXAnimationController::AdvanceTime function ultimately controls all animation processing. If the animation set is not registered with the animation controller, the animation controller will have no knowledge of the animation set and will not use it. Therefore, we can think of this function as returning an animation set that is currently a detached object (i.e., not yet plugged into the animation subsystem). We will shortly discuss how to manually create an ID3DXAnimationController and how to register animation sets with it, but for now you can rest assured that when we use D3DXLoadMeshHierarchyFromX and specify an X file that contains animation data, all of this will be handled for us automatically.

Note as well that when D3DX loads an X file that contains multiple animation sets, although all animation sets will be registered with the returned animation controller, only the last animation set defined in the X file will be assigned to an active track on the animation mixer (track 0). This is why we do not have to configure the animation controller passed back to our application in Lab Project 10.1. The X file in that project contains only a single animation set and it is automatically assigned to track 0 on the animation mixer by the D3DX loading function. Thus it is ready for immediate playback. If an X file contains multiple animation sets, and the last animation set is not the first one you would like to playback, you can use the ID3DXAnimationController interface to assign a different animation set to that track (or to another enabled track if you intend to do animation blending).

Manually Populating the Animation Set

After a keyframed animation set has been manually created, it is initially empty and contains no SRT data. The next task of the application will usually be to add the SRT data for each frame in the hierarchy that this animation set intends to animate. We call the ID3DXKeyframedAnimationSet::RegisterAnimationSRTKeys method to place the SRT data for a single frame in the hierarchy into the animation set. In other words, if you wanted your animation set to animate five frames in your hierarchy (five animations), then you would call this method five times. Each call would specify the SRT data for a single frame. The method is shown below with a description of each of its parameters.

```
HRESULT RegisterAnimationSRTKeys
(
    LPCSTR pName,
    UINT NumScaleKeys,
    UINT NumRotationKeys,
    UINT NumTranslationKeys,
    CONST LPD3DXKEY_VECTOR3 *pScaleKeys,
    CONST LPD3DXKEY_QUATERNION *pRotationKeys,
    CONST LPD3DXKEY_VECTOR3 *pTranslationKeys,
    DWORD *pAnimationIndex
);
```

LPCSTR pName
This parameter is how we inform the animation set which frame in the hierarchy the keyframe data we are passing in is intended for. This string should contain the name of a frame in the hierarchy and thus

this also becomes the name of the animation itself. This is the animation name you would use in calls to methods like ID3DXAnimationSet::GetAnimationIndexByName if you wished to retrieve the index of the animation within the animation set's internal animation array.

UINT NumScaleKeys
UINT NumRotationKeys
UINT NumTranslationKeys
These three parameters (any of which may be zero) inform the animation set how many separate scale, rotation and translation keyframes we are going to register with the animation set for this animation. This allows the function to allocate the internal SRT arrays of the animation to the requested size. It also informs the function how many keys to expect in the arrays we pass in using the next three parameters.

CONST LPD3DXKEY_VECTOR3 ***pScaleKeys***
CONST LPD3DXKEY_QUATERNION ***pRotationKeys***
CONST LPD3DXKEY_VECTOR3 ***pTranslationKeys***
These three parameters are used to pass in the already allocated and populated arrays of the scale, rotation and translation keyframes. The application should have allocated these arrays and populated them with keyframe data prior to registering them with the animation set. The animation set will copy these keyframes into its internal SRT arrays and thus, these arrays may be released from memory after the function has returned. Any of these pointers can be set to NULL, indicating that there is no keyframe data of that type. For example, if the pScaleKeys parameter is set to NULL then this animation does not contain any scale animation data. You should also make sure that if this is the case, the NumScaleKeys parameter is also set to zero.

DWORD ***pAnimationIndex***
For the final parameter we can pass in the address of a DWORD which on function return will contain the index of the animation we have just created. The application may wish to store this away and use it later to access the animation keys without having to first call GetAnimationIndexByName to retrieve the index first. You may pass NULL as this parameter if you do not wish to store the index of the animation about to be created.

To better understand the creation process, the following example code will create an ID3DXKeyframedAnimationSet and will add animation data to it for a frame in the hierarchy called 'MyFrame'. In this simple example, the animation data will contain just two scale keys at timestamps of 0 and 2000 ticks. This animation set will be created with a ticks per second ratio of 100. This means the first keyframe will be executed at 0 seconds and the second keyframe will be executed at 20 seconds (2000 / 100).

```
// Create new keyframed anim set called "Anim1" with a 100m ticks per second
// It is looping, will contain 1 animation and will have 0 callback keys (0,NULL)
ID3DXKeyframedAnimationSet * pAnimSet = NULL;
D3DXCreateKeyFramedAnimationSet("Anim1",100,D3DXPLAY_LOOP, 1, 0, NULL, &pAnimSet);

// Let us now allocate an array of two scale keys. At time zero the frame matrix
// is scaled by 5 and at time 2000 ( 20 seconds ) the matrix is scaled 10x
D3DXKEY_VECTOR3 ScaleKeys[2];
ScaleKeys[0].Time = 0;
ScaleKeys[0].Value= D3DXVECTOR3( 5 , 5 , 5);
ScaleKeys[1].Time = 2000;
```

```
ScaleKeys[1].Value= D3DXVECTOR3( 10 , 10 , 10);

// Now add a new animation to the animation set for 'MyFrame'. It has:
// 2 scale key, 0 rotation keys, 0 translation keys. We pass in the ScaleKeys
// array and pass NULL for the other two key types. We also pass in NULL as the
// animation index which we do not require in this example.
pAnimSet->RegisterAnimationSRTKeys("MyFrame",2,0,0,ScaleKeys,NULL,NULL,NULL);
```

At this point we have an animation set with keyframe data for a single frame in the hierarchy but it is still not plugged into the D3DX animation system. For that to happen we must register the animation set with the animation controller (covered shortly). Once that is done, we can assign it to tracks on the animation mixer and play it back in response to calls to ID3DXAnimatonController::AdvanceTime by the application.

In the next section we will finally discuss the ID3DXAnimationController interface. While it exposes a great many methods (thus making it very flexible), do not feel intimidated. We have already learned that using the animation controller can be very straightforward for basic animation tasks (see Lab Project 10.1). At its simplest, we can call a single method (AdvanceTime) to play our animations. However, we can do many more exciting things with this powerful interface, so let us begin to explore how it works.

10.7 The ID3DXAnimationController Interface

As discussed in the last section, there may be times when D3DXLoadMeshHierarchyFromX is not an option available to us for loading animation data. In those cases, we will have to tackle things manually. Just as there are D3DX functions to manually create keyframed animation sets, there is also a global D3DX function that allows us to create an animation controller to manage them. This function is called D3DXCreateAnimationController. Examining the parameter list to this function will help us to better understand the various internal components of animation controller objects and how they work together. So we will break with tradition and cover the manual creation of the animation controller first. But before we cover this function and its parameters, let us first briefly review, at a high level, what the D3DXAnimationController is and what its responsibilities are in the animation system. This is the interface we are returned from D3DXLoadMeshHierarchyFromX and often is the only interface in the D3DX animation system that our application will work with directly. You are reminded that you do not need to create an animation controller manually if you are loading the data from an X file (D3DX creates the animation controller for you and returns it from the loading function). However there may be times when the animation controller created by D3DXLoadMeshHierarchyFromX is not suitable for our application's needs. In such cases, we will need to know how to create or clone a new controller.

Animation Set Manager

Just as the ID3DXKeyframedAnimationSet can be thought of as an animation manager, the ID3DXAnimationController can be thought of as an ID3DXAnimationSet manager. The

ID3DXAnimationController interface can be used to assign any of its currently managed animation sets to tracks on its internal animation mixer. These tracks can be played back individually or blended together to create complex animation sequences. Each track on the animation mixer has several properties which can be set (speed, position, weight, etc.) to influence the way a given animation set assigned to the track plays back at runtime.

Manually creating an animation controller starts us off with an empty container (much like manually creating an animation set). On its own it is not much use because it does not currently have any animation data assigned to it for management purposes. So the first thing we will usually do after creating an animation controller is call its RegisterAnimationSet method to register various (already created) animation sets. Once an animation set has been registered with the animation controller, it will need to be assigned to a mixer track if we intend to use it during playback to influence the matrices in our frame hierarchy. From there, any calls to AdvanceTime will cause the animation set to update the appropriate nodes in the hierarchy as defined by the artist/animator.

Animation Set Mixer

Animation set registration does not activate the animation -- the animation set must be assigned to an active mixer track for the pre-recorded sequence to be played. When an application calls ID3DXAnimationController::AdvanceTime, only currently active tracks on the animation mixer will have their periodic positions updated. Actually, it is the track the animation set is assigned to which has its own local time which is updated and not the animation set itself. It is the track's time that is used to generate the periodic position of the animation set which in turn is used to generate the SRT data. Therefore, when we call the AdvanceTime method of the animation controller, this elapsed time is also added to the local clocks of each track. By using the track time to generate the periodic position of its assigned animation set, we have the ability to forward or rewind the timeline of individual animation sets on their tracks.

While an animation controller might have many animation sets registered with it, it might currently have only one set assigned to an active track. This would be the only set used to update the hierarchy during the AdvanceTime call. This approach allows the application to play different animation sets at different times simply by assigning animation sets to active tracks when they need to be played. If we have multiple animation sets assigned to active tracks on the mixer, all of those animation sets will affect their respective frames in the hierarchy simultaneously. The final transformation for a given frame in the hierarchy will be the blended result from each animation in each active animation set that is linked to that specific frame.

The ID3DXAnimationController interface has many functions that govern how animation sets are assigned to the various tracks on the mixer and how much weight those sets will contribute to the final blended result. Animation set weighting allows us to play, for example, two animations that might both manipulate some of the same frames in the hierarchy, but allows one set to have more influence over the final result than the other. For example, the first animation set might influence a given frame at 75% of its full weight while the second set only influences the same frame at 25% of its full weight. So while both animation sets have influenced the hierarchy, there was a 3:1 influence ratio in favor of the first set (thus the animations stored in set one would be more visually dominant than those in set two).

Frame Matrix Updates (Animation Outputs)

We have said on a number of occasions that the animation controller uses its animations to manipulate frames in the hierarchy. In truth, the animation controller is really only concerned with the frame matrices themselves for a given named animation. In truth, we could connect them to any arbitrary matrices we choose -- not necessarily only matrices in a frame hierarchy. We will see momentarily that the animation controller maintains a list of pointers to all the frame matrices in the hierarchy that it needs to animate. Internally these matrix pointers are stored in an array so that the animation controller can quickly access them when it needs to rebuild them (when AdvanceTime is called). These matrices that will receive the final SRT output from the mixer are referred to as "Animation Outputs" by D3DX. Thus, as we will see in the next section when we build an animation controller manually, in addition to registering all of the animation sets with the controller, we must also register all of the output matrices in the hierarchy that the animation controller will need to update, so that it has access to those matrix pointers. If a frame's matrix is not registered with the animation controller, then it cannot be manipulated by any of its registered animations (even though those animations are linked to frames in the hierarchy by name). In fact, when we register a matrix with the animation controller, we must also give it a name. This matrix name must match the name of the animation that will be used to animate that frame.

Of course, all of this is handled on our behalf if we use D3DXLoadMeshHierarchyFromX. D3DX will assign each animation within an animation set the name of the frame that it animates (if one exists) and the matrix of that frame will also be registered with the animation controller using the same name as the animation. If we have a frame in our hierarchy called 'RotorBlades', then when D3DX creates the animations(s) that animate this frame, they too will be assigned the name 'RotorBlades'. Furthermore, the transformation matrix of that frame will be registered as an animation output in the animation controller's matrix table and assigned the name 'RotorBlades' as well. This clearly establishes the relationship between the matrix and the animation which alters its value with animation results.

When we manually create our own animation controllers, we will register frame matrices using the ID3DXAnimationController::RegisterAnimationOutput function. It is important to note however that because of the nature of hierarchical data structures (any changes to a matrix will affect all child frame matrices) we must register *all* matrices that the animation controller needs to rebuild. This means that in our example, we must not only register the transformation matrix of our RotorBlade frame, but all child frame matrices of the RotorBlade frame as well. This allows the animation controller to correctly update all affected matrices. While this sounds rather tedious, fortunately there is a global D3DX function to handle this matrix registration with a single function call -- it is called D3DXFrameRegisterNamedMatrices and we will look at it a little later. Basically, we just pass this function a pointer to the root frame of our hierarchy and a pointer to our animation controller interface, and it will step through the hierarchy and register all of the matrices that belong to 'named' frames in the hierarchy. Thus we can manually register all of our matrices with the animation controller with a single function call.

> **Note:** D3DXLoadMeshHierarchyFromX will automatically create the animation controller and register all animation sets and matrices. This means that the returned animation controller interface can be used immediately for hierarchy animation as seen in Lab Project 10.1.

You might have a concern about the dependency of the system on names, given our earlier discussion about the optional nature of frame names in X files. Not to worry however. While it is true that an X file frame data object may not be assigned a name, this will not be the case if that frame is being animated. Remember that inside the X file the Animation data object (i.e. keyframe data) is linked to a frame by embedding the frame's name inside the Animation object as a data reference object. If the frame did not have a name, it could not be bound to any animation inside the X file either. So we can be confident that animated frames within the X file will always have assigned names. D3DX will thus have what it needs to register both the animation assigned to a frame and its matrix with the animation controller, to sustain the link between them.

Memory Management

Animation controllers have a number of restrictions that can be imposed on them by the application or by D3DX at creation time. These limitations allow us to create more memory-friendly animation controllers that conserve space. For example, if we only plan to use one animation set at a time, then we know that we only need to use one track on the animation mixer. It would be wasteful if the mixer was always created with 128 tracks for example. Likewise, it would be wasteful for the animation controller to be created with a matrix pointer array large enough to animate a maximum of 1000 matrices if we only wish to animate a single matrix in our hierarchy. The same is true with regards to how many animation sets must be set up for management. If we know that we will only register four animation sets in total with the controller, it would be wasteful to create an animation controller that was capable (at the cost of some internal memory overhead) of managing 20 animation sets.

While these figures are arbitrary, the important point is that when an animation controller is created manually (using the D3DXCreateAnimationController function), such size restrictions are set at this point and remain fixed for the lifetime of the controller. In the case of the D3DXLoadMeshHierarchyFromX function, D3DX will create the animation controller for us, so setting these limitations is out of our hands. We will see in a moment exactly what the limitations of the animation controller created by D3DX are when we cover manually creating one.

While these restrictions are fixed for a given controller after creation, the ID3DXAnimationController does include a Clone function which allows us to create a new animation controller from an existing one (such as the one D3DX created on our behalf). Much like we saw in Chapter Eight when discussing the cloning of meshes, when we clone a new controller we will be able to extend the capabilities of the original controller. Therefore, if the capabilities of the animation controller that D3DXLoadMeshHierarchyFromX created for us did not meet our requirements, we can clone the animation controller into a version that does. The new controller would retain all of the registered animation sets and keyframe data, so we do not have to rebuild or re-register them. This allows us to make a copy of the animation controller but extend its features (ex. increase the maximum number of tracks or maximum number of matrices, etc.) much as we did with meshes in Chapter Eight.

Sequencing

The animation controller also implements an event scheduler referred to as the sequencer. Using methods exposed by the ID3DXAnimationController interface, we can set events on the controller's *global* timeline which modify the properties of a specific mixer track (or its clock) when that event is triggered. When we register events with the sequencer, we must make sure that we understand which timeline those events are being scheduled for. For example, we know that every time we call AdvanceTime, the elapsed time passed into the function is added to the global time of the animation controller. Not an awful lot happens with the global time of the animation controller if sequencing is not being used. Essentially it is just a running total of elapsed time. To be clear, this global time is *not* the time used to map to a periodic position in an animation set. That is the track time. We discussed earlier in this chapter that when we call AdvanceTime and the global time of the controller is updated, the same elapsed time is also added to the timers of each active track on the mixer. It is the timer/clock of a track that is used to map to a periodic position in its animation set prior to any GetSRT calls.

It may seem strange at first for the controller to maintain a global timer and a timer for each track when essentially the same elapsed time passed into the AdvanceTime function is added to all of the timers simultaneously. If this is the case, then it would seem that the global time of the controller and the timers of each track would always have the same time as one another. After all, they all start off at zero and have the same elapsed time added to them every time AdvanceTime is called by the application. While this is certainly true if we are performing nothing more complex than simple AdvanceTime calls, we will see that we can decouple the global timer from the track timers by adjusting the properties of each track. The sequencer allows us to schedule events in global time that will alter the properties of given track or even its timer. For example, we could set an event that is triggered at a global time of 10 seconds which disables a specific track and its assigned animation set. We could also set another event at 20 seconds (global time) that would re-enable that same track. At this point, the timer of the track would be different from the global timer because it was not being updated along with the global timer after it was disabled. Thus, in this scenario, when the global time has accumulated to 25 seconds, the track (which was disabled for 10 seconds) would have a timer that contains 15 seconds of accumulated time. Again, it is the 15 second track time that would be mapped into a periodic position and used to fetch the SRT data of animations in its assigned animation set.

Another example demonstrating that sequencer events are added to the global timeline is one in which we intend to alter the timer of a track at a specific global time. For example, imagine we have an animation set with a period of 20 seconds. Let us also imagine that we set an event on the sequencer at 20 seconds that will set the timer of the track back to zero. Let us also assume that this animation set is not configured for looping, that is to say, it is designed to play through only once.

In this scenario, the application would call AdvanceTime repeatedly, thus updating both the global time of the controller and the track time of the animation set by the same amount. At 20 seconds global time, an event is found on the global timeline that alters the track timer of our animation set back to zero. At this point, the track time would be zero and the global time would be 20. At 30 seconds global time, the animation set would be 10 seconds into its second iteration even though the animation set was not configured to loop. This is because we manually set the track timer back to zero at a global time of 20 seconds, which starts that track playing from the beginning again. As far at that animation set is

concerned, it is like it is playing for the first time. What would happen at a global time of 40 seconds? Would the track, having reached the end of its period for a second time, loop back to the beginning again? No, it would not. We have already said that the animation is not set to loop and as we have no event set on the sequencer for 40 seconds, no action is taken. At 41 seconds, the track time will be set at 21 seconds. When the track time of 21 is mapped to the periodic position of the animation set, it is found to have exceeded its period. Since the animation set is not configured to loop, at 21 seconds (track time) the animation has finished.

Finally, imagine that we had also scheduled an event that reset the track time back to zero again at a global time of 100 seconds. When a global time of 100 seconds was reached, the animation would start to run from the beginning again (once) for the next twenty seconds, until the track timer once again exceeded 20 seconds (the period of the non-looping animation set). Therefore, for the global time period 100-120 seconds, the track timer was once again in the range 0-20 seconds.

Other possible sequencing events include the ability to alter the playback speed of a given animation set at a particular moment, and the ability to dynamically change a mixer track blending weight at a certain global time so that the animation set assigned to that track can contribute more or less to the final result at the given timestamp.

> **Note**: Sequencer events are registered with the controller's global timeline. Even if an animation set is set to loop or ping-pong, the event will only ever be triggered once, when the global time passes the event on the timeline. This is true even if we reset the track timer. If we had an event scheduled at 20 seconds on the global timeline which reset the timer of a track to zero, the event would not be triggered again when the same track time reaches 20 seconds for a second time. This is because we are only resetting the track time and not the global time. The global time continues to increase with each call to AdvanceTime and would at this point be at 40 seconds. We will see in a moment how we can also reset the global time.

As with all other capabilities, when we create an animation controller, it will have a maximum number defining how many sequencing events can be simultaneously set. When we create the animation controller manually, we can set this value according to the needs of our application. But when using D3DXLoadMeshHierarchyFromX (where the animation controller is created on our behalf), the maximum will be 30 key events scheduled at any one time. If we need to schedule more key events than this, then we will have to clone the animation controller into one with greater event management capabilities. Do not worry; using the sequencer will be explained in detail momentarily. Just know for now that it exists to provide us the ability to script changes for mixer tracks at specific global times.

10.7.1 D3DXCreateAnimationController

Now that we have a basic idea of what the animation controller has to offer, we can begin to examine the specifics. While our application will not normally need to create an animation controller from scratch (because D3DXLoadMeshHierarchyFromX will do it for us), it helps to examine the process since it provides good insight into the internal data layout and inner workings of the animation controller. Remember that if you are loading your animation data from a custom format and cannot use the D3DXLoadMeshHierarchyFromX function, you will also need to manually build your keyframe animations and animation sets in addition to the animation controller used to playback that data.

The D3DX library provides a global function called D3DXCreateAnimationController for creating animation controllers. This function takes several parameters which define the limitations of the newly created animation controller. The function is shown below, followed by a discussion of its parameters.

```
HRESULT D3DXCreateAnimationController
(
      UINT MaxNumAnimationOutputs,
      UINT MaxNumAnimationSets,
      UINT MaxNumTracks,
      UINT MaxNumEvents,
      LPD3DXANIMATIONCONTROLLER *ppAnimController
);
```

UINT MaxNumAnimationOutputs

This parameter represents the maximum number of hierarchy frame matrices that the animation controller can manipulate during a single `AdvanceTime` call. By default, when our animation has been loaded via `D3DXLoadFrameHierarchyFromX`, this value will be set to the number of frames in the hierarchy that will be animated by the data stored in the file, including any child frames they may own. D3DX will register all of these matrices with the animation controller on our behalf when using D3DXLoadMeshHierarchyFromX.

To register additional matrices with the animation controller returned from D3DXLoadMeshHierarchyFromX we will need to clone the animation controller and adjust the MaxNumAnimationOutputs setting to provide space for those additional matrices. Individual matrix registration can be accomplished via the ID3DXAnimationController::RegisterAnimationOutput function, which we will study in a short while. This function will not generally be of concern to us when using D3DXLoadMeshHierarchyFromX, but it is needed for manual controller creation.

UINT MaxNumAnimationSets

This parameter describes the maximum number of animation sets that can be managed (registered via its RegisterAnimationSet method) at any one time by the animation controller. When loading the animation from an X file using D3DXLoadMeshHierarchyFromX, the default value is the total number of animation sets actually stored in the file. To register additional animation sets with the animation controller, we must clone a new one and update this value according to our needs. As mentioned earlier, many of the popular X file exporters currently support exporting only a single animation set. Thus the animation controller created will only provide storage for a single animation set to be registered. We can determine how many sets are registered with the controller using its GetMaxNumAnimationSets method.

UINT MaxNumTracks

This is the maximum number of mixer tracks that the controller should simultaneously support. Essentially it represents the maximum number of animation sets that can be simultaneously assigned to mixer tracks for blending. By default, this value equals 2 when we load animation data from an X file via D3DXLoadMeshHierarchyFromX. If we need additional tracks, we will simply clone a new controller to support however many animation mixer tracks are required.

Note that when we clone a new controller to add tracks, we can specify as many new tracks as we want, but each track will occupy roughly 56 bytes of memory (before data is assigned to it). Again, the

number of available tracks determines the maximum number of animation sets that can be simultaneously blended. Of course, we may have many more registered animation sets than available tracks on our animation mixer. For example, we might have 6 animation sets that should be played back (and blended) in groups of two, such that sets 1 and 2 might be played together, sets 3 and 4 might be played together and sets 5 and 6 might be played together. In this situation we would want to create the animation controller with a 'MaxNumTracks' of 2 but a 'MaxNumAnimationSets' of 6 since we need to manage six different animation sets but only need to playback two at a time.

UINT MaxNumEvents
This parameter allows us to specify the maximum number of key events that can be registered with the animation controller's sequencer. The animation controller sequencer is analogous to a MIDI sequencer used for music composition. Those of you that have worked with MIDI sequencer applications should recall that the sequencer allows for MIDI events to be placed on a given MIDI channel. These MIDI events might include instructions to change the keyboard instrument being used to play that track at a certain time or change the volume of a certain instrument at a certain time. With regards to animation sequencing, this feature affords us the ability to schedule special events to be triggered at certain points in the global timeline of the animation controller. These events include changing the playback speed of a specified track, modifying track blend weights, or jumping to/from locations on a track's local timeline at a specified global time. We will cover the sequencing functions a little later in the chapter.

The default animation controller created by D3DXLoadMeshHierarchyFromX is capable of storing a maximum of 30 key events. If this is not suitable, then we can clone the animation controller to extend the sequencing capabilities.

LPD3DXANIMATIONCONTROLLER *ppAnimController
The final parameter to this function is the address of a pointer to an ID3DXAnimationController interface. On successful function return, it will point to a valid interface for a D3DXAnimationController object.

10.7.2 ID3DXAnimation Controller Setup

The animation controller returned via the manual creation method discussed in the last section is initially empty. The first thing we will want to do is register all of the matrices we need the animation controller to animate, along with any animation sets we have created. While this will not be the case when using D3DXLoadMeshHierarchyFromX, we will need to understand the process when managing our own custom data.

Registering Frame Matrices

While it would not be difficult to write code that recursively steps through the hierarchy and registers any animated frame matrices, D3DX exposes a global function that will do this for us. It is called D3DXFrameRegisterNamedMatrices and is shown below along with a description of its parameters. We will pass this function a pointer to a D3DXFRAME structure from our hierarchy and it will register that

frame's matrix, along with any child frame matrices, with the animation controller passed in as the second parameter. Usually we will pass in a pointer to the hierarchy root frame so that all frame matrices (that have names) are registered with the animation controller.

```
HRESULT D3DXFrameRegisterNamedMatrices
(
    LPD3DXFRAME pFrameRoot,
    LPD3DXANIMATIONCONTROLLER pAnimController
);
```

LPD3DXFRAME pFrameRoot

This is the top level frame where hierarchy traversal will start. The function will register this frame's matrix (if it is named) and subsequently traverse its children, registering any named matrices it encounters. Passing the hierarchy root frame will ensure that all frame matrices that can be animated in the hierarchy are properly registered with the animation controller.

LPD3DXANIMATIONCONTROLLER pAnimController

The second parameter is a pointer to the animation controller that we would like the hierarchy matrices registered with.

While this function makes our job easy, there is one small issue which we must resolve. Recall that during animation controller creation, we will need to specify the maximum number of matrices that the controller will manage (MaxNumAnimationOutputs). As a result, we will need to know the count of named frame matrices in our hierarchy *before* the controller is created. While we could write our own traversal counting routine, D3DX again steps up to make our job easier by providing a global function called D3DXFrameNumNamedMatrices.

```
UINT D3DXFrameNumNamedMatrices
(
     LPD3DXFRAME pFrameRoot
);
```

The function takes a single parameter -- a pointer to a frame in our hierarchy (usually the root frame). It will return the number of named frames (including the passed frame) that exist in the subtree, including all named children of the passed frame. We can use the value returned to create the animation controller and then register the matrices using the previously discussed function.

Example

The following code snippet creates an animation controller and registers the matrices of the hierarchy. In this example it is assumed that pRoot is a D3DXFRAME pointer to the root frame of an already created hierarchy.

```
ID3DXAnimationController * pAnimController = NULL;
UINT MaxNumMatrices = D3DXFrameNumNamedMatrices ( pRoot );
D3DXCreateAnimationController ( MaxNumMatrices , 4 , 2 , 10 , &pAnimController );
D3DXFrameRegisterNamedMatrices ( pRoot , pAnimController );
```

The example builds an animation controller that is capable of managing four animation sets. It has an animation mixer with two channels (which means it is capable of blending only two animation sets at once) and can have ten key events set in its internal sequencer.

> **Note:** We remind you once again that these steps are only required when we manually create an animation controller. D3DXLoadMeshHierarchyFromX does all of this work on our behalf before returning a fully functional animation controller interface.

Registering Animation Sets

Once we have created our animation controller and registered all matrices that will be animated by our animation sets, it is time to register those animation sets with the animation controller. Until we do, the animation controller has no animation data (it has is a list of matrices, but no idea about how to fill them with animation results).

For each animation set that needs to be registered with the animation controller we call the ID3DXAnimationController::RegisterAnimationSet method. It accepts a single parameter -- a pointer to a valid ID3DXAnimationSet interface. By 'valid' we mean that it should already contain all of its keyframe data. This function would be called once for each animation set to be registered with the animation controller.

```
HRESULT RegisterAnimationSet( LPD3DXANIMATIONSET pAnimSet );
```

> **Note:** When an animation set is being registered, the controller calls the animation set's GetAnimationIndexByName function for each frame. Its goal is to retrieve either a valid index via the pIndex output parameter and D3D_OK, or D3DERR_NOTFOUND via the function result. Caching the animation indices bypasses the need to look up the index on the fly. It also tells the controller exactly which frames even have any animation data (those which resulted in D3DERR_NOTFOUND do not have a matching animation). Once this process is completed, AddRef is called to indicate the controller's use of the animation set that has just been registered.

It is important to realize that the number of animation sets we register with the controller must not be greater than the MaxNumAnimationSets parameter passed into the D3DXCreateAnimationController function. If you did not create the animation controller yourself and would like to know the maximum number of animation sets that can be registered with the animation controller, you can use the ID3DXAnimationController::GetMaxNumAnimationSets method of the interface as shown below. The function returns the total number of animation sets that can be registered with the animation controller simultaneously.

```
UINT GetMaxNumAnimationSets(VOID);
```

Assuming that we have previously created four animation sets and wish to register them with our animation controller, the code might look something along the lines of the following. The next code snippet assumes that pRoot points to a valid D3DXFRAME structure, which is the root frame of our hierarchy. It also assumes the four animation sets (and their interpolators) have already been created.

```
ID3DXAnimationController *pAnimController = NULL;
ID3DXAnimationSet *pAnimSet1= ‘ Valid Animation Set’
ID3DXAnimationSet *pAnimSet2= ‘ Valid Animation Set’
ID3DXAnimationSet *pAnimSet3= ‘ Valid Animation Set’
ID3DXAnimationSet *pAnimSet4= ‘ Valid Animation Set’

// Create animation controller
UINT MaxNumMatrices = D3DXFrameNumNamedMatrices ( pRoot );
D3DXCreateAnimationController ( MaxNumMatrices , 4 , 2 , 10 , &pAnimController );

// Register Matrices
D3DXFrameRegisterNamedMatrices ( pRoot , pAnimController );

// Register Animation Sets
pAnimController->RegisterAnimationSet ( pAnimSet1 );
pAnimController->RegisterAnimationSet ( pAnimSet2 );
pAnimController->RegisterAnimationSet ( pAnimSet3 );
pAnimController->RegisterAnimationSet ( pAnimSet4 );
```

At this point the animation controller will have all of the data (e.g. keyframe data and the matrices they manipulate) it needs to animate the hierarchy. However, no animation sets have yet been assigned to tracks on the mixer. This last step is something that will need to be done before we call AdvanceTime. We will discuss mixer track assignment in a moment.

Our application can retrieve the current number of animation sets that have been registered with the controller using the ID3DXAnimationController::GetNumAnimationSets method. This allows us to query how many of the animation set slots allocated for the controller are already occupied with animation sets. We can only register another animation set with the controller if:

```
pController->GetMaxNumAnimationSets() > pController->GetNumAnimationSets()
```

The function that retrieves the current total of registered animation sets is:

`UINT GetNumAnimationSets(VOID);`

Unregistering an Animation Set

Just as we can register animation sets with the animation controller, we can also unregister them. This might be useful if you have decided that you are finished with an animation set and will not need it again in the future. In this case you can unregister the set and then release its interface. It is also useful when you have more animation sets than your current animation controller can handle. Let us say for example that you had 10 animation sets which you planned on using but for some reason your animation controller was created to support only five registered sets at any one time. In this case you could register animation sets when they are needed and then unregister them when they are no longer needed to make room for other animation sets. The function is shown below and takes a single parameter -- a pointer to the ID3DXAnimationSet you wish to unregister.

`HRESULT UnregisterAnimationSet(LPD3DXANIMATIONSET pAnimSet);`

Note that to remove the animation set we must pass in the interface pointer. This function will call Release on the animation set when it relinquishes control (assuming that it owns the animation set we requested to be unregistered). So if we wanted to unregister an already existing animation set being managed by the controller that we do not currently have a pointer to, we would need to get access to it first. The following code uses the GetAnimationSet method of the ID3DXAnimationController interface (which we have not covered yet) to retrieve the ID3DXAnimationSet assigned to slot 3 in the animation controller's animation set array. We then unregister the animation set and finally release it.

```
LPD3DXANIMATIONSET pAnimSet;

// Fetch interface to 4th registered animation set
m_pAnimController->GetAnimationSet( 3, &pAnimSet );

// At this point, the animation set 'pAnimSet',
// will have a reference count of '2'.
m_pAnimController->UnregisterAnimationSet( pAnimSet );

// Release has been called, reference count is now '1'
pAnimSet->Release(); // Animation set deleted.
```

In the above code you can see that we retrieved the fourth animation set being managed by the controller (the one at index 3). Now that we have unregistered it, passing 3 to GetAnimationSet on a subsequent call will return a pointer to the animation set which used to exist at index 4. In other words, unregistering an animation set will shuffle all of the other animation sets that follow it back one slot in the animation controller's internal animation set array. It is also worth noting that any unregistered animation set will be removed from the mixer track if it is currently assigned to one. Any calls to GetTrackAnimationSet for the track which previously owned it will now return NULL until another animation set is assigned to that track.

When we call ID3DXAnimationController::Release, all of the registered animation sets are unregistered and their reference counts decremented. Therefore, we do not have to explicitly call UnregisterAnimationSet for each registered animation set before we release the animation controller interface.

> **Note:** When using D3DXLoadMeshHierarchyFromX to load animation data from an X file, all animation sets contained in the X file are created and registered with the animation controller on our behalf. There is no need to call the above function unless you need to register additional animation sets (not contained in the X file) with the animation controller. In that case you will probably need to clone the animation controller to make room for your extra animation sets.

Registering Matrices

As discussed previously, D3DXFrameRegisterNamedMatrices does all of the hard work of registering the matrices in a given hierarchy with an animation controller. This function is basically just traversing the hierarchy that we pass it and calling the ID3DXAnimationController::RegisterAnimationOutput method of the animation controller interface for each named matrix that it finds. If you need to manually

register some or all of your matrices with the animation controller, you can do so yourself using this method.

```
HRESULT RegisterAnimationOutput
(
    LPCSTR Name,
    D3DXMATRIX *pMatrix,
    D3DXVECTOR3 *pScale,
    D3DXQUATERNION *pRotation,
    D3DXVECTOR3 *pTranslation
);
```

LPCSTR Name
The first parameter is the name that you would like the output to be registered under. This name should match the name of an animation in an animation set.

D3DXMATRIX *pMatrix
The matrix pointer passed into this function is stored by the animation controller so that it can update the frame matrix with SRT data without traversing the hierarchy each time. When we use the D3DXLoadMeshHierarchyFromX function to load an X file which has animation data, this function will be called multiple times to store the matrix of each named frame in the hierarchy in the controller's matrix pointer table. While this may sound complicated, bear in mind that all we are doing is informing the controller about a matrix (usually the matrix of a D3DXFRAME structure) that is to receive the SRT data from animations with the passed name. This allows the controller to build a matrix pointer array for quick updates of the frame hierarchy matrices.

D3DXVECTOR3 *_pScale_
D3DXQUATERNION *_pRotation_
D3DXVECTOR3 *_pTranslation_
This function was updated in the 2003 summer update of the SDK and now NULL can be passed as the matrix parameter and the SRT data stored as separate components. This allows an application to store the SRT data from a given animation in either a matrix, or have it stored in a separate scale vector, a rotation quaternion and a translation vector instead. While an automatically created animation controller (one created by D3DXLoadMeshHierarchyFromX) will have this function called for each named frame with the final three parameters set to NULL, and will only register the frame matrix as an animation output, this does provide applications with the flexibility to store the SRT data output from a given animation in separate variables. For example, an application performing its own animation blending that still wishes to use the animation controller for SRT generation (keyframe interpolation) might decide not to have the output stored in a matrix but as a separate scale, rotation and translation that it can later blend into a matrix using its own custom techniques. If any parameter (except the first) is set to NULL, the associated SRT data will not be output. For example, if the pMatrix, pScale and pTranslation parameters are set to NULL and only a rotation quaternion pointer is registered, then only the rotation portion of the SRT data will be output. When using the full D3DX animation system, it is not likely that you will ever want the pMatrix parameter set to NULL, as this is usually a pointer to the matrix of a frame in the hierarchy that you wish to be updated when AdvanceTime is called. It is often the case that both the matrix will be registered as an animation output along with some or all of the final three parameters. This allows the animation controller to update the frame in the hierarchy (pMatrix) and also return SRT data to the application, which may be used to calculate physics for example.

It is important that you do not register more outputs with the animation controller than the maximum number allowed. The maximum number of output that can be registered with the controller is set when the animation controller is created (the MaxNumAnimationOutputs parameter) using the D3DXCreateAnimationController function. You can retrieve the maximum number of supported outputs by calling the ID3DXAnimationController::GetMaxNumAnimationOutputs function.

```
UINT GetMaxNumAnimationOutputs(VOID);
```

This function returns an unsigned integer value describing the maximum number of outputs that can be registered with the animation controller at any one time. As a result, this value also tells us the maximum number of frame matrices that can be simultaneously animated by the controller. A single output can be a combination of a matrix, a scale and translation vector and a rotation quaternion.

We have now covered how to create and configure the animation controller, and how to register all required matrices and animation sets. So the animation controller now has all of the data it needs for animating our hierarchy matrices. Again, we will rarely need to perform any of these tasks if we are simply loading animated X files, because D3DXLoadMeshHierarchyFromX will do the work for us.

The next step is to understand how to use the methods of the ID3DXAnimationController interface to configure the animation mixer. This will allow us to start experimenting with how our animation sets are played back in our application.

10.8 The Animation Mixer

Although we have registered our animation sets with the animation controller (or it has been done for us by D3DXLoadMeshHierarchyFromX), we are not quite ready yet to start playing them back. Earlier we learned that we will use the ID3DXAnimationController::AdvanceTime method to progress along its global timeline and animate the hierarchy. While this is certainly the case, keep in mind that we might have multiple animation sets registered with the animation controller and that we will want the ability to play back only a subset of those animation sets at any given time. For example, if we had an animated character hierarchy with three animation sets (‘Walk’, ‘Swim’, ‘Die’), we will probably want to play back only one of these sets at any given time.

The animation mixer allows us to select the animation set(s) we wish to use for animating the hierarchy each time AdvanceTime is called. It also allows us to have multiple animation sets assigned to different tracks so that they can be blended together to produce more complex animations.

When the animation controller is first created, it is at this time that we determine the maximum number of mixing tracks that the animation mixer will be able to use. This lets us scale the animation mixer to suit our needs so that memory is not wasted by maintaining tracks that our application will never use. You should recall from our discussion of the D3DXCreateAnimationController function that the MaxNumTracks parameter defines the maximum number of tracks for the animation controller. To use the analogy of an audio multi-track mixing desk once again, these mixing desks come in different shapes and sizes. The more tracks an audio mixing desk has, the more individual sounds we can simultaneously mix together. For example, guitar, bass, drums, and vocals are generally assigned their own separate tracks when your band is recording an album. This allows the studio sound engineer to isolate and tweak each individual instrument sound for better results (or to add effects, etc.). Although each track maintains only the data for that particular instrument, all tracks will be played back simultaneously on the final album, blended together to produce the song in its complete form.

With the animation mixer, the number of tracks defines the maximum number of animations sets that can be blended together at any one time. So in order to play an animation set, we must first assign it to a track on the animation mixer and then enable that track. Tracks can be enabled and disabled by our application at will, so we have total control over the process in that respect. Next we will discuss how to assign animation sets that have been registered with the animation controller to tracks on the animation mixer. This way they will be ready for use the next time our application calls the ID3DXAnimationController::AdvanceTime function.

10.8.1 Assigning Animation Sets to Mixer Tracks

When an animation controller object is *manually* created and its animation sets are registered, by default none of these animation sets will be assigned to any tracks on the animation mixer. Before we play an animation we must select the appropriate animation set(s) and assign them to the mixer track(s) we wish to use. This is analogous to the painter who has many jars of colored paints in his workshop, but will only put a few blobs of paint on his palette to use in the current painting. We can think of the registered animation sets as the available jars of paint. Some of these sets will be placed into the mixer (the palette) to be used to create the final animation (the painting). Note that the artist may decide to smear some of the paint blobs together on his palette to create new color blends for the painting. We can do the same thing with our animation mixer.

Assuming that we have registered our animation sets with the animation controller, we can assign any animation set to a valid track on the animation mixer. By a ‘valid’ track, we simply mean a track with an index in the range [0, MaxNumTracks - 1]. Unlike the DirectX lighting module, where we can assign lights to any arbitrary lighting slot index just so long as we do not enable too many lights simultaneously, animation mixer tracks are always zero-based and consecutive in their numbering scheme. Therefore, if we created an animation controller that supports a maximum of eight mixer tracks, this means that we can assign a different (registered) animation set to tracks 0 through 7 on the

animation mixer. Note that the index of an animation set is not related to the index of the track to which it can be assigned. The index of an animation set is merely representative of the order in which it was registered with the controller. It is perfectly acceptable to assign animation set 5 to animation mixer track 2 for example.

Keep in mind that when the animation controller is created during the D3DXLoadMeshHierarchyFromX call, all animation sets will be registered, but only the last animation set defined in the file will be assigned to a track (track 0, specifically). By default, the animation controller will support a maximum of two tracks and the second track will be disabled with no assigned animation set. If you require additional tracks to produce your animations, you must clone the animation controller to extend its capabilities.

We assign an animation set to a mixer track on the animation mixer using the ID3DXAnimationController::SetTrackAnimationSet function. The first parameter is the zero-based number of the track we want the animation set assigned to. The second parameter is the interface pointer of the animation set being assigned.

```
HRESULT SetTrackAnimationSet( DWORD Track,
                              LPD3DXANIMATIONSET pAnimSet );
```

The animation set passed in must already be registered with the controller. Any previous animation set already assigned to the specified track will be unassigned before the new animation set assignment takes place.

If you do not have the actual interface of the animation set handy, but you know the index of the animation set (based on when it was registered) then you can use the ID3DXAnimationController::GetAnimationSet function to retrieve its interface.

```
HRESULT GetAnimationSet( DWORD iAnimationSet,
                         LPD3DXANIMATIONSET *ppAnimSet );
```

We pass this function the index of the registered animation set interface we want to retrieve and the address of an animation set interface pointer which will store the returned result. This function will increment the reference count of the object before returning the interface, so be sure to release it when you have finished with it.

Next we see some example code that acquires an interface pointer for the third registered animation set (AnimationSet[2]) and assigns it to track 7 on the animation mixer. This code assumes that the animation controller was created with a minimum of an 8-track animation mixer.

```
// Set up track 0 for the animation mixer
LPD3DXANIMATIONSET pAnimSet;
m_pAnimController->GetAnimationSet( 2, &pAnimSet );
m_pAnimController->SetTrackAnimationSet( 7, pAnimSet );
pAnimSet->Release();
```

It is much more intuitive to find animation sets by name (ex. “Jump”) and as we might expect, we can also retrieve the animation set interface by name also. This function, which is shown next, accepts a string containing the name of the animation we are searching for and the address of a pointer to an ID3DXAnimationSet interface. On successful function return this will point to a valid animation set.

```
HRESULT GetAnimationSetByName( LPCSTR pName,
                               LPD3DXANIMATIONSET *ppAnimSet);
```

In the following example, we show how we might assign an animation set called “Shoot” to mixer track 3 (i.e., the fourth track).

```
// Set up track 0 for the animation mixer
LPD3DXANIMATIONSET pAnimSet;
m_pAnimController->GetAnimationSetByName( “Shoot”, &pAnimSet );
m_pAnimController->SetTrackAnimationSet( 3, pAnimSet );
pAnimSet->Release();
```

So as you can see, we have two useful ways to retrieve the interface for a registered animation set that we wish to assign to a track. We can retrieve its interface by name or by index.

```
HRESULT GetTrackAnimationSet( DWORD Track,
                              LPD3DXANIMATIONSET *ppAnimSet );
```

ID3DXAnimationController exposes another animation set retrieval function called GetTrackAnimationSet which allows us to specify a track number and get the interface for the animation set currently assigned to that track. This is handy if we wish to fetch the animation set assigned to a given track on the mixer and assign it to another track. It is also useful if we want to unregister the animation set because we have no further use for it. Note that unregistering an animation set will also remove its track assignment.

The function takes two parameters -- the first is the index of the mixer track containing the animation set we wish to obtain an interface for, and the second is the address of an ID3DXAnimationSet interface pointer. The latter parameter will point to a valid interface to the animation set on function return. If no animation set is assigned to the specified track, the interface pointer will be set to NULL.

Once again, we must remember to release the interface when we are finished with it so that the underlying COM object will be properly cleaned up when it is no longer needed by any modules.

The following code shows how we might assign three animation sets (sets 2, 5, and 8) to animation mixer tracks 0, 1, and 2. It assumes that the animation controller has already been created with a minimum of 9 pre-registered animation sets and at least a 3-track mixer. This example will be typical of code that might be executed in response to a certain action or event taking place in our game logic. For example, imagine that this particular animation controller is managing the hierarchy of an animated character and that the animation sets are mapped as follows:

Set 2 = Walk
Set 5 = Fire Weapon
Set 8 = Act Wounded

If the character was currently experiencing all of these states then we could assign them to the first three tracks of the animation mixer. Then the next time we call AdvanceTime, the character will be animated using the blended result of these three animations sets. As a result the character would fire his weapon while walking along, behaving like he has sustained injuries. This next example will assume that the PlayerAction variable is a DWORD that has bits set that our application identifies with the walking, firing, and wounded action states.

```
if ((PlayerAction & WALK) && (PlayerAction & FIRING) && (PlayerAction & WOUNDED) )
{
      LPD3DXANIMATIONSET pAnimSet2 = NULL;
      LPD3DXANIMATIONSET pAnimSet5 = NULL;
      LPD3DXANIMATIONSET pAnimSet8 = NULL;

      m_pAnimController->GetAnimationSet( 2, &pAnimSet2 );
      m_pAnimController->GetAnimationSet( 5, &pAnimSet5);
      m_pAnimController->GetAnimationSet( 8, &pAnimSet8);

      m_pAnimController->SetTrackAnimationSet( 0, pAnimSet2 );
      m_pAnimController->SetTrackAnimationSet( 1, pAnimSet5 );
      m_pAnimController->SetTrackAnimationSet( 2, pAnimSet8 );

      pAnimSet2->Release();
      pAnimSet5->Release();
      pAnimSet8->Release();
}
```

The code block above sets the animations we want to play on this animation controller (at least until the character enters another action mode). When we call ID3DXAnimationController::AdvanceTime in our render loop, these animations can then be blended together for our final on-screen result. However, this is not all we will usually have to do. After all, we need to find the right balance between these three animations to make sure that the results are correct. Mixing animations is more an art than a science, and we will often have to experiment a bit to find good blends that work for us. Fortunately, each track on the animation mixer has several properties that can be set to control how the animation applied to that track is played out and how that track is blended with animation sets assigned to other tracks.

In the next section we will examine how to set up the individual tracks of the animation mixer so that they play their assigned animations in the desired way.

10.8.2 Animation Mixer Track Configuration

Every animation mixer track has five configurable properties which determine the behavior of the track and thus the animation set assigned to it.

The first property **enables/disables** the track. If a track is disabled it will not be used in the AdvanceTime call regardless of whether it contains an animation set. Being able to enable/disable tracks on the fly is a useful feature in a game. Imagine that we have two animation sets assigned to our mixer: track one stores a ‘Walk’ animation which we repeatedly loop as the character navigates the world. On

the second track we store a short animation called ‘Wave’ which makes the character wave his hand to other characters in the world in greeting as he passes them by. Since we would only want the second animation to play when the character spots a friend, we would disable this track by default. At this point, every call to AdvanceTime would only update the hierarchy using the ‘Walk’ animation set assigned to track one. But once the character spots a buddy, we could enable track two for a set period of time (a few seconds perhaps) before disabling it again. In the few seconds that track two is enabled, any calls to AdvanceTime would blend the two animation sets together such that the character model would be walking and waving at the same time.

The next important mixer track property is **weight**. Going back to our audio mixing desk analogy, we can think of the weight of a track as being akin to the volume knob. Increasing the volume of a track will make the instrument assigned to that track louder in the final mix. The weight property of the animation track is similarly a way to scale the influence of its assigned animation set on the hierarchy. The track weighting value is usually in the range [0.0, 1.0] where a value of 1.0 means the animation set assigned to that track manipulates the hierarchy at its full strength. A value of 0.0 means the animation set will not influence the hierarchy at all. This value can be set higher than 1.0 although the resulting hierarchy transformation might not be what you expect. We will look at a track’s primary weight value in more detail in a moment.

The third configurable property of a track is its **speed**. Speed is a scalar value which allows us to define how fast we want the animation set to be played back. A speed value of 1.0 means ‘normal speed’ (i.e. the speed that it would play at by default as intended by the animation creator). Specifying a value of 2.0 for instance means the animation set assigned to the track would play out twice as fast. This might sound like a strange feature to have, but it is actually nice to have the flexibility to play the animation sets back at different speeds (imagine that you wanted to do a slow-motion sequence where all animations ran at half time). Setting the speed of a track is another feature that allows us to decouple the global time of the controller from the timer of a given track. We can think of the speed parameter of a track as being a scalar that is multiplied by the elapsed time passed into the Advance Time call prior to being added to the track timer. As an example, let us imagine that we pass into AdvanceTime an elapsed time of 2 seconds. We know that the global timer of the controller would have two seconds added to it. However, before this elapsed time is also added to the timer of each track, it is scaled by the speed parameter of that track. If the speed of the track was set to 0.25, then the actual elapsed time added to the track would be ElapsedTime*0.25 = 0.5. As you can see, although the global time and the rest of the animation tracks have moved along their time lines by 2 seconds, the track in question has only had its timer updated by ½ a second. Therefore, the animation set assigned to this track appears to be playing back in slow motion compared to other sets being played. In this example, it is playing back at ¼ of the speed of the global timer. Obviously, speed values greater than 1 (the default) will make the animation set play back faster than global time. For example, a speed value of 2 will cause the animation to play back at twice its speed.

The fourth property of a mixer track is a time value called ***position***, which is essentially the value of the timer for that track. Therefore, in DirectX terminology, the global time is the time of the animation controller which is the sum of total elapsed time, and the position is the current value of a single track timer (which itself is the sum of all elapsed time passed onto that track during updates). The ***periodic position*** is the value of the track ***position*** mapped into the period of the animation set, taking into account looping and ping-ponging. It is the track position that is passed into the

ID3DXAnimationSet::GetPeriodicPosition function to generate an animation set local time that is ultimately passed on to the ID3DXAnimationSet::GetSRT function.

To understand the three layers of time used by the D3DX animation system, below we show a simplified pseudo-code version of how the ID3DXAnimationController::AdvanceTime function would apply updates.

```
AdvanceTime(  ElapsedTime )
{
    // Add elapsed time to the animation controllers global time
    GlobalTime += ElapsedTime;

    // Loop though each enabled track
    For (Each Enabled Track)
    {
       // Update the position of the track timer scaling by track speed
       Track->Position += (ElapsedTime * Track->Speed)

       // Fetch animation set assigned to current track
       AnimationSet = Track->GetTrackAnimationSet( )

       // Map the track timer ( position ) into a periodic position for the set
       PeriodicPositon =AnimationSet->GetPeriodicPosition( Track->Position )

       // Use periodic position to generate SRT data for each animation in this set
        For ( Each Animation In Animation Set)
        {
              AnimationSet->GetSRT( PeriodicPosition , Animation ,
                                    &scale , &rot , &trans )

          .. Do other stuff with SRT data here …
        } // Each Animation

     } // End Enabled Track
}
```

We will usually want to set the position of a track to zero before the animation set is played for the first time, so that the animation set assigned to that track starts at the beginning.

While we will often have no need to interfere with the local time (the position) of a mixer track and will usually allow the animation controller to advance them on our behalf when we call AdvanceTime, we do have the ability to override this behavior and offset a track's animation set to a specific track local time. Imagine that we initially assigned two animation sets to tracks 1 and 2. We could set track 1's position to 0 and track 2's position to 5. When we start playing our animation using AdvanceTime, track 1 would start playing its animation set from the beginning while track 2 would start playing from 5 seconds in (assuming the period of the animation was at least 5 seconds long). This is probably a more useful feature if your animation set is actually a container filled with multiple animations (as is often the case for the default export of X files). In such a case your engine would need to know where particular animation routines started and ended in the set (ex. 'Run' animation from 0 seconds to 2 seconds, 'Shoot' animation from second 3 to second 6, etc.) and pass these times appropriately. It is a bit more complicated of course because your engine will also need to control when the animation should stop and

be reset to play again from the beginning (which is why splitting up animation sets according to individual animation routines makes life much easier – as we will do in Lab Project 10.2).

The fifth and final property that we can assign to a mixer track is a **priority** value. With it, each track can be assigned one of two possible priorities: high or low. Low priority tracks are actually blended together separately from high priority tracks. Later, the two blended results are blended together using the animation mixer's **priority blend weight** (which is another weight we can set). For example, assume that we have an 8-track mixer and we assign the first four tracks a low priority and the second four tracks a high priority. When we call AdvanceTime, the low priority tracks (0-3) would be blended together using their respective per-track weights and the high priority tracks (4-7) would be blended together using their respective per-track weights. Finally, the results of both blends for a given frame are blended together using the priority blend weight set for the mixer. This is a single property value for the entire animation mixer that we can configure as needed. If it was set to 0.25 for example, it means that after the low priority tracks have been blended (we will call this Result 1) and the high priority tracks had been blended (we will call this Result 2), the final transformations applied to the hierarchy would be a combination of 25% of Result1 and 75% of Result2. This provides another layer of animation blending control beyond the per-track blend weights. Figure 9.8 gives us a graphical idea of the concepts just discussed. It demonstrates an 8-track mixer with all of the configurable properties outlined previously, along with the input (animation sets) and output (transformations).

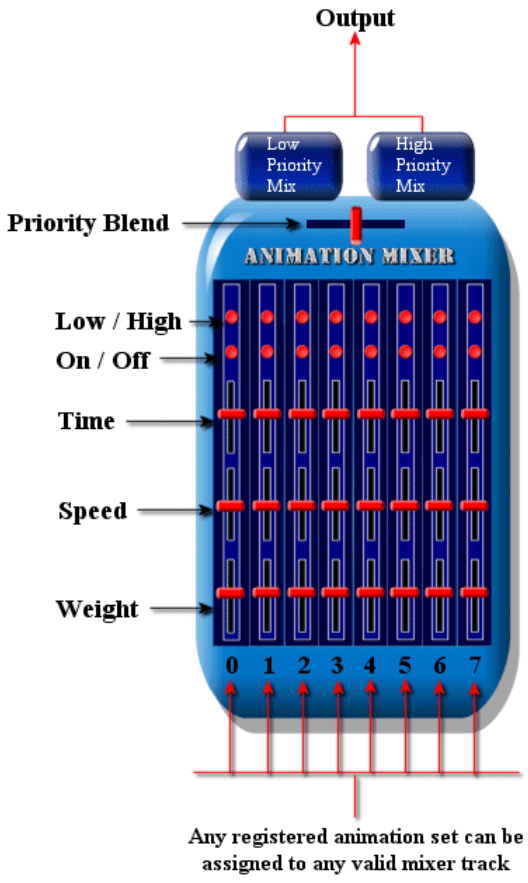


Figure 9.8

Enabling / Disabling a Track

The first thing we will usually want to do when we assign an animation set to a track is make sure that track is enabled. This will ensure that the track and its assigned set are used in the next call to AdvanceTime. We enable and disable tracks using the ID3DXAnimationController::SetTrackEnable method.

```
HRESULT SetTrackEnable( UINT Track, BOOL Enable );
```

The first parameter to this function is the index of the mixer track we wish to enable or disable. The second parameter is either TRUE or FALSE depending on whether we would like to enable or disable the mixer track respectively. To mix a track with other tracks, this flag must be set to TRUE. Conversely, setting this flag to FALSE will prevent the specified track from being mixed with other tracks. You should not leave tracks enabled if they have no animation sets assigned to them.

Setting Track Weight

Setting the weight of a track controls the influence of the animation set assigned to it (i.e. how strongly it manipulates the hierarchy). We can set the per-track weight value using the ID3DXAnimationController::SetTrackWeight function.

```
HRESULT SetTrackWeight( UINT Track, FLOAT Weight );
```

The first parameter is the index of the track we are specifying the weight for and the second parameter is the floating point weight value itself. This is the primary track weighting value and it will generally be specified in the [0.0, 1.0] range. Although you can specify values outside of this range if it serves your purpose, there is the potential for some strange results. All weights for all active tracks are essentially normalized anyway so if you have only one track and set its weight to 0.5, it will essentially have a weight of 1.0 anyway. The controller makes sure that the sum of all track weights equals one prior to the blending process.

Let us imagine that we have a single cube in our scene hierarchy, and that we have two separate animation sets that will manipulate that cube. The first animation set translates the cube 100 units to the right and the second set translates the cube 100 units to the left. If both of these sets were placed in separate tracks, and both had a weight of 1.0, nothing would actually happen to the cube. Each set would produce translations which counteract one another:

```
(100 * 1.0) + (-100 * 1.0) = 0
```

However if we were to use a weight of 0.4 for the first set (which moves the cube 100 units to the right) the cube itself will have moved 60 units to the left by the end of the animation:

```
(100 * 0.4) + (-100 * 1.0) = -60
```

While this example is very simple, you can certainly understand the basic principles involved here. This is a generic blending formula that we have seen many times before (in Chapter Seven for example). In a game, we might use the weight value for blending an animation set of a character firing a gun (with only the associated animations applied to the arms), with a recoil animation set that we can blend at different strengths based on the type of weapon being fired.

It is worth noting that, by definition, the weight value can be used to define the strength of an animation, even in cases where hierarchy frames are not being manipulated by more than one track. Although keep in mind that it will have no effect on the animation. For example, if we had two tracks and neither manipulated the same hierarchy frames as the other one, the weight would not scale the strength of the animation applied to the respective matrices. You will probably set your per-track weights to 1.0 most of the time so that each animation set affects the hierarchy in the exact way intended by the creator of the animation data. This is certainly true if your animation sets generally influence their own unique portions of the hierarchy.

As discussed previously, the per-track weights are blended with other tracks assigned to the same priority group (high or low). The results of each priority group are then blended together using the priority blend weight of the animation controller to produce the final SRT data for a given frame. Therefore, even if a track is assigned a weight of 0.5, it does not necessarily mean that it will influence the hierarchy at half strength. The final strength of influence may be reduced or boosted depending on how the two priority groups are combined in the final weighting stage.

Setting Track Speed

Recall that while we pass an elapsed time (in seconds) to the controller's AdvanceTime method to update the global timer of the controller, each track maintains it own local timer which is also updated with the elapsed time passed in. The current track time is referred to as the track's position. For each track we can set a speed value (a single scalar) that is essentially used to scale the elapsed time we pass in and thus the amount that we wish to increment the timer of each track. If this scalar is set to 1.0 (the default) then the animation set will play back at its default speed -- the speed intended by the animation creator -- and the elapsed time passed into the AdvanceTime function will be added directly to the timer of the track without modification. Alternatively, if we set this value to 0.5, then the animation set assigned to that track would play out at only half the speed. This is because the elapsed time is scaled by the track speed before being added to the track's timer. A track with a speed setting of 0.5 will progress at half the speed of the global timeline of the controller. If a track is initially set with a speed value of 0.25, then at twenty seconds of global time, the track's timer will be at 5 seconds. As it is the track's timer that is mapped to a periodic position and used to generate SRT data, this directly effects the speed at which the animation set assigned to the track plays out.

```
HRESULT SetTrackSpeed( UINT Track, FLOAT Speed );
```

The first parameter should be the index of the track you wish to set the speed for. The second parameter is a floating point value indicating the desired speed scale value.

Setting Track Animation Position

As discussed earlier, each track maintains its own local timer. Every time the AdvanceTime method is called and the global timer is updated, so too are the timers for each track. The elapsed time passed in will be scaled by the speed of a track before being used to increment the track timeline. The current value of a track's timer is referred to as its position.

Not only do we have the ability to control the speed at which a track's counter updates using the elapsed time and its speed, we also have the ability to set the track timer to any arbitrary position. This allows us to cause the animation set to jump to a specific local time at a specific global time.

Usually, we will set the initial position for each track to zero before we play any animations, to ensure that all of the animation sets start from the beginning. Nevertheless, we can call the ID3DXAnimationController::SetTrackPosition function whenever we wish to set (or offset) the local timer of a specific track to a specific track position. We could, for example, have an animation set that is looping but needs to always restart from the beginning every time an event happens in the game, regardless of its current periodic position.

```
HRESULT SetTrackPosition( UINT Track, DOUBLE Position );
```

The first parameter is the track we wish to set the time for and the second parameter is the position we wish to set the track to. The next time we call the AdvanceTime method, all additions to the timeline will be relative to this new track position. As it is the track time that is mapped into a periodic position in the animation set, setting the position of a track directly effects the periodic position also.

The next code snippet demonstrates how we might want to set position of the mixer to start from the beginning (regardless of where this track may currently be in its timeline) in response to the 'FireWeapon' command. This will cause the animation set assigned to that track to restart regardless of whether it is configured to loop or not. It is assumed when looking at this code that the fTime parameter contains the elapsed time (in seconds) that has passed since the last frame. This is the value that will be added to the global time of the controller and used to increment the timer of each track (scaled by track speed). It is also assumed that there are multiple animation sets assigned to tracks on the animation mixer and that these animation sets will loop when they reach their end. When the fire button is pressed however, the animation set assigned to track 2 on the mixer restarts because we set the track time to zero, effectively forcing a restart of just that animation set. This will not affect any of the other animation sets assigned to other tracks; they will continue to play out as normal without any knowledge of the restart in track 2.

```
void UpdateAnimations ( double fTime , ID3DXAnimationController * pController)
{
      if (FireButton)
            pController->SetTrackPosition ( 2 , 0.0f );

      // Set the new  global time
      pController->AdvanceTime ( fTime );
}
```

It is worth remembering that although we set the track timer to zero, we then proceed to call the AdvanceTime method passing in the advanced time. Therefore, the next time the SRT data for this animation set if generated, the position of the track will in fact be 0 + ElapsedTime, as we would fully expect. This track time would be mapped to the periodic position of the animation set and used to fetch the two bounding keyframes used for interpolation.

> **Note:** Remember, the AdvanceTime method updates the matrices of the individual hierarchy frames. It is only when you traverse the hierarchy and combine the newly updated matrices that you will be ready to draw the final results.

Setting Track Priority

This function allows us to assign each track to one of two blending groups: a high priority group or a low priority group. Each group will have its tracks blended together separately using the per-track weight and eventually the blended results for each group will be blended into a final result used to update the hierarchy matrices. As we have the ability to alter the way in which the two priority blend groups are combined (using the controller Priority Blend Weight which we will discuss in a moment) this provides us a second layer of blend control.

This method is shown below and as you can see it takes two parameters. The first is the track index and the second is a member of the D3DXPRIORITY_TYPE enumeration.

```
HRESULT SetTrackPriority( UINT Track, D3DXPRIORITY_TYPE Priority);
```

The D3DXPRIORITY_TYPE enumeration is defined by D3DX as:

```
typedef enum _D3DXPRIORITY_TYPE
{
   D3DXPRIORITY_LOW = 0,
   D3DXPRIORITY_HIGH = 1,
   D3DXEDT_FORCE_DWORD = 0x7fffffff
} D3DXPRIORITY_TYPE;
```

As you can see there are two choices (D3DXPRIORITY_LOW or D3DXPRIORITY_HIGH) indicating which of the two possible groups the track should be assigned to. Naming these groups High Priority and Low Priority is perhaps a little misleading as it incorrectly suggests that tracks assigned to the high priority groups will always have a higher influence on the hierarchy than those in the low priority groups. In actual fact, we will see in a moment how the animation controller lets us set a priority blend weight for the entire controller (a value between zero and one) which allows us to control how the high and low priority blend groups are blended. A value of 0.5 for example would blend both groups with equal weight. Alternatively, this value could also be set such that the low priority group has a greater influence. Rather than thinking of these groups as high and low priority, it is perhaps easier to think of them as being just two containers which can have arbitrary priority assigned to them. We will see how to set the priority blend weight of the animation controller in a moment.

Setting Track Descriptors

We have now seen how to set a track's speed, weight, priority group and position as well as how to enable or disable the track at will. The functions we have looked at are typically going to be called while our animation is playing so that we can adjust the track properties as needed, perhaps in response to some game state change. As it turns out, we can also setup these values (typically when we first create the animation controller) using a single function call. This function is called ID3DXAnimationController::SetTrackDesc and it allows us to set the speed, weight, priority groups and position of a track with a single call. The method is shown below.

```
HRESULT SetTrackDesc( UINT Track, D3DXTRACK_DESC *pDesc );
```

The first parameter is the number of the track we wish to set the properties for and the second parameter is the address of a D3DXTRACK_DESC structure containing the values for each property. The structure is shown below followed by a description of each member. The meaning of each member should be obvious to you given the functions we have covered in the previous section.

```
typedef struct _D3DXTRACK_DESC
{
    D3DXPRIORITY_TYPE Priority;
    FLOAT Weight;
    FLOAT Speed;
    DOUBLE Position;
    BOOL Enable;
} D3DXTRACK_DESC, *LPD3DXTRACK_DESC;
```

DWORD D3DXPRIORITY_TYPE
This parameter should be set to either `D3DXPRIORITY_LOW` or `D3DXPRIORITY_HIGH`. This allows us to assign the track to one of two blend groups: a high priority group or a low priority group as discussed earlier. Each group will have its tracks blended together separately and eventually the blended results for each group will be blended into a final result used to update the hierarchy matrices.

FLOAT Weight
This value has exactly the same meaning as the 'Weight' parameter we pass to the SetTrackWeight method. It allows us to set the current weight of the track which determines how much influence the assigned animation set has on the hierarchy.

FLOAT Speed
This member has exactly the same meaning as the 'Speed' parameter in the SetTrackSpeed function. It allows us to set a scalar value that is used by the controller to scale the animation time update for a given animation set. A value of 1.0 will run the animation at its default speed while a value of 2.0 will run the animation set at twice the default speed, and so on.

DOUBLE Position
This member has exactly the same meaning as the 'Position' parameter in the SetTrackPosition function. It allows us to set the timer of a track to a specific position without affecting any other tracks.

As a simple example, this allows us to restart the animation set assigned to a single track on the mixer without affecting other tracks.

BOOL Enable
This member is a Boolean variable where we pass in 'TRUE' or 'FALSE' to enable or disable the track, respectively.

This single function can be used instead of all the standalone functions we discussed in the previous sections. Whether you are initialising the properties of each track or simply changing the properties of a track midway through your game, this function provides a nice way of assigning multiple properties to a track with a single function call.

Finally, our application can retrieve the properties of a given track by using the ID3DXAnimationController::GetTrackDesc function.

```
HRESULT GetTrackDesc( DWORD Track, D3DXTRACK_DESC *pDesc );
```

The first parameter should be the track index we wish to retrieve the property information for and the second parameter should be the address of a D3DXTRACK_DESC structure to be filled with results.

Priority Blending

Each mixer track can be assigned to either a low priority group or a high priority group. The tracks of each group are blended together and then the resulting transformations for each group are blended together using what is referred to as the *priority blend weight* of the animation mixer. This is a global setting for the animation mixer and it controls how the low priority blend result is combined with the high priority blend result. We use the ID3DXAnimationController::SetPriorityBlend function to set this weight as a floating point value between 0.0 and 1.0.

```
HRESULT SetPriorityBlend ( FLOAT BlendWeight );
```

Tracks are assigned to priority groups using the either the SetTrackPriority function or the SetTrackDesc function discussed in the last section. Group assignment is based on one of the following flags:

D3DXPRIORITY_LOW – Track should be blended with all other low priority tracks ***before*** mixing with the high-priority result.

D3DXPRIORITY_HIGH – Track should be blended with all other high priority tracks, ***before*** mixing with the low-priority result.

Let us return to an earlier example to get some more insight into this concept. Recall that we have two tracks: one translates 100 units to the right and the other translates 100 units to the left. When each had a per-track weight of 1.0 we ended up with no movement because they cancelled one another out. However, if we were to specify them uniquely as low/high priority tracks, we see the following:

```
Track1: ( 100 * 1.0 ) = 100       // High priority track
Track2: ( -100 * 1.0 ) = -100     // Low priority track

// First Test
BlendWeight = 0.25 // One quarter high, three quarters low

Output = Track1 + ((Track2 - Track1) * BlendWeight)
Output = 100 + ((-100 - 100) * 0.25) = 50
Output = One quarter from 100 (high) and three quarters from -100 (low)

// Next Test
BlendWeight = 0.5       // Half way between low and high
Output = Track1 + ((Track2 - Track1 ) * BlendWeight)
Output = 100 + ((-100 - 100) * 0.5) = 0
Output = Half way between -100 (high) and 100 (low)

// Last Test
BlendWeight = 0.75      // Three quarters high, one quarter low

Output = Track1 + ((Track2 - Track1) * BlendWeight)
Output = 100 + ((-100 - 100) * 0.75) = -50
Output = Three quarters from 100 (high) and one quarter from -100 (low)
```

You can see that this is really just a simple linear interpolation between high and low priority tracks.

Our application can retrieve the current priority blend mode of the animation mixer by calling the ID3DXAnimationController::GetPriorityBlend function. It returns a single floating point value as a result.

```
FLOAT GetPriorityBlend( VOID );
```

10.8.3 Animation Mixer Configuration Summary

You should now be comfortable with the notion of assigning animation sets to tracks on the animation mixer and modifying track properties to effect the blended animations that are output. The following code snippet demonstrates how we might assign animation sets to two tracks in the animation controller and configure them in different ways. This code assumes that the animation controller has already had its animation sets registered.

```
pAnimController->SetTrackAnimationSet ( 0 , pAnimSetA );
pAnimController->SetTrackAnimationSet ( 1 , pAnimSetB );

D3DXTRACK_DESC Td;
Td.Priority =     D3DXPRIORITY_HIGH;      // Assign High Priority Group
Td.Enable   =     TRUE;                   // Enable Track
Td.Weight   =     1.0f;                   // Set Default Weight
Td.Speed    =     1.0f;                   // Set Default Speed
Td.AnimTime =     0.0f;                   // Start animation set at beginning
pAnimController->SetTrackDesc ( 0, &Td );
```

```
Td.Flags    =     D3DXPRIORITY_LOW ;      // Assign Low Priority
Td.Enable   =     TRUE;                   // Enable Track
Td.Weight   =     1.0f;                   // Set Default Weight
Td.Speed    =     0.5f;                   // Play track 1 at half its actual speed
Td.AnimTime =     12.0f;                  // Start this animation set 12 seconds in
pAnimController->SetTrackDesc ( 1, &Td );

// We have now set the track properties.
// Let us set the priority blend weight to distribute blending
// between high and low priority groups equally (a 50:50 blend)
pAnimController->SetPriorityBlend ( 0.5f );
```

Over time, you will find lots of creative ways to make use of animation mixing in your applications. Perhaps in one application you will find that you need only per-track weighting to achieve your desired results and you will forego using the priority blending system. In another application you may decide that you will mix two types of animation methods together (ex. keyframe animation with inverse kinematics) and find that the priority blending system works well for you. Either way, we have certainly seen thus far that D3DX provides quite a sophisticated system that, in the end, can save us a good deal of coding time.

10.8.4 Cloning the Animation Controller

Earlier in the chapter we learned that during the creation of our animation controller we will specify certain limitations such as the maximum number matrices or maximum number of animation sets that will be managed. While this presents no problem when we manually create an animation controller, because we can set the limits the animation controller to suit our needs, we might find ourselves disappointed with the default restrictions specified by D3DXLoadMeshHierarchyFromX when we load our animation data from an X file. The animation controller returned from D3DXLoadMeshHierarchyFromX has the following default limits:

MaxNumMatrices = Number of animated matrices in the X file hierarchy
MaxNumAnimSets = Number of animation sets in the X file (usually 1)
MaxNumTracks = 2
MaxNumEvents = 30

The one limitation that may be the hardest to accept is the two-track animation mixer which restricts us to blending only two simultaneous animations. It may also be problematic (as logical as the design decision was) to limit ourselves to the number of animation sets defined in the X file being the maximum number that can be registered. For example, although the X file might only contain a single animation set, we may want to generate another animation set programmatically and register that with the controller too. We would not be able to do so under the current circumstances, as this would exceed the maximum number of animation tracks that can be registered with the controller.

Fortunately, we are not locked in to the defaults that D3DXLoadMeshHierarchyFromX imposes. Just as we are able to clone a mesh to extend its capabilities (Chapter Eight), we can also clone an animation controller and modify its capabilities. Cloning creates a new animation controller with the desired

capabilities while preserving the animation, matrix, and animation set data of the original controller. That data will be copied into the newly generated controller such that it is ready to animate our hierarchy immediately. Once we have cloned the animation controller, the original animation controller can be released.

We clone an animation controller using the ID3DXAnimationController::CloneAnimationController function shown next. It is nearly identical to the D3DXCreateAnimationController function, so the parameter list should require no explanation.

```
HRESULT CloneAnimationController
(
    UINT MaxNumAnimationOutputs,
    UINT MaxNumAnimationSets,
    UINT MaxNumTracks,
    UINT MaxNumEvents,
    LPD3DXANIMATIONCONTROLLER *ppAnimController
);
```

> **Note:** We will cover events (parameter four) later in this chapter when we cover the animation sequencer.

10.8.5 Hierarchy Animation with ID3DXAnimationController

Once we have the animation controller that suits our needs, we are ready to use it to animate our hierarchy. While we have looked at quite a few functions for setup and configuration for both our animation controller and animation mixer, once all of that is done, actually playing the animations could not be easier. In fact, there is really just a single function (AdvanceTime) that we need to use.

AdvanceTime takes as its first parameter the time (in seconds) that has elapsed since the last call to AdvanceTime (which will be zero the first time we call it) and then updates the internal timer of the controller and the timer of each animation track (scaled by track speed). It then maps the timer of each track into the periodic position of its assigned animation set and uses this periodic position to call the animation set's GetSRT method for each animation it contains to generate the new SRT data for the frame to which it is attached. If multiple tracks (animation sets) are being used, then it is possible that there may be an animation in each animation set that animates the same frame in the hierarchy. In that situation we know that there will be multiple sets of SRT data for that single hierarchy frame. The animation mixer will blend those SRT sets together to calculate the final transformation matrix for that frame based on the properties we discussed earlier. The AdvanceTime function is shown next. Please note that this function takes a second optional parameter which allows us to pass in a pointer to a callback function. We will discuss the callback feature of the animation system at the end of this chapter, so for now just assume that we are passing in NULL as this parameter.

```
HRESULT AdvanceTime( DOUBLE Time,
                     LPD3DXANIMATIONCALLBACKHANDLER pCallbackHandler);
```

Simply put, once the animation controller has been configured, playing animation involves nothing more than calling this function in your game update loop, passing in the time that has elapsed since the previous call. The next code example uses our timer class to get the number of seconds that have passed since the last call and passes it into the AdvanceTime method of the animation controller. After this call, all the parent-relative matrices in the hierarchy will have been updated.

```
void MyClass::UpdateAnimations ( )
{
      double ElapsedTime =  m_Timer.GetTimeElapsed() ;
      m_pAnimController->AdvanceTime( ElapsedTime );
}
```

This single function call will calculate all of the current animation data and update the hierarchy frame matrices so that our meshes can be scaled, rotated, and translated according to the animation being played. Once the above function has been called, the hierarchy is ready to be traversed (where we will calculate the world matrices) and rendered as discussed earlier. Our approach to rendering the hierarchy is not altered at all by the fact that we are animating it. The animation controller simply changed the values stored in the relative matrices of our frame hierarchy so that the next time it is updated and rendered the meshes appear in their new positions.

If your animations are set to loop, you do not have to worry about keeping a running total of the sum of elapsed time or wrapping back round to zero after exceeding the length of the longest running animation set. This is because once the track timers are updated, they are mapped into the periodic position of the assigned animation set. If an animation set is configured to loop or ping-pong, then regardless of how high the value of the track's timer climbs, it will always be mapped into a periodic position within the bounding keyframes of the animation set.

If your animation sets are not configured for continuous play, then some care will need to be taken when calling AdvanceTime. Once a track timer exceeds the period of its animation set, the animation set will cease to play (technically, it has ended at this point). Of course, we have seen that you have the ability to set the position of a track back to zero to cause this animation set to play again, even if the global time exceeds the length of the animation set. It is helpful that the global time is decoupled from the track timers in this way.

As an example, imagine that we have an animation set with a period of 15 seconds which is not set to loop. It also has a track speed of 1.0, so initially there is a 1:1 mapping between global time, track time and the periodic position of the animation set. We might decide to perform out own looping logic so that we can control when the animation set starts playing. Imagine that although the animation set has a period of 15 seconds, we only want it to start playing again at a global time of 20 seconds, thus causing 5 seconds of inactivity between the time the first animation loop ends and the second one begins. The following code would achieve this (note that it assumes that the animation set is assigned to track 0 on the mixer):

```
void ProcessAnimations ( double ElapsedTime )
{
      static double WrapTime  = 20.0;

      double GlobalTime = pController->GetTime();
```

```
      if ( GlobalTime >= WrapTime )
      {
            pController->SetTrackPosition( 0 , GlobalTime-WrapTime );
            WrapTime+=20.0f
      }

      pController->AdvanceTime ( ElapsedTime );
};
```

Notice the call to the controller's GetTime method. This is a new method which we have not yet discussed. GetTime returns the global time of the controller. Remember, the global time is really the sum of all elapsed times that have been passed into our AdvanceTime calls since the animation began.

DOUBLE GetTime(VOID);

The previous code initially sets the wrap around time at 20 seconds global time. When this time is reached, the track timer is wrapped around to zero and a new wrap around time of 40 seconds is set. At 40 seconds the track timer will be wrapped around to zero and a new global wrap around time of 60 seconds would be set, and so on. While the period of the animation set is only 15 seconds and is not configured to loop, the above code would reset the track timer to 0 at every 20 second global time boundary. This clearly shows the distinction between the global timer which is ever increasing with each AdvanceTime call, and the track timers which can be reset, positioned or stopped independent of the global timer.

For no other reason than to better understand this concept, let us take a quick look at how we might write a simplified AdvanceTime function of our own that works with multiple animation sets. This will let us see how the timers are incremented. We will not worry about animation mixing in this example and will assume that each animation set influences its own section of then hierarchy. This will give us a rough idea of what is happening behind the scenes in the D3DXAnimationController::AdvanceTime function. Obviously there is a lot missing here, mainly the mixing of multiple frame SRT sets and the callback and sequencing systems which we will discuss shortly. However, this example does allow us to get the basic idea of how the update works and how the controller interacts with each track and its assigned animation sets. Fortunately, we will never have to write a function like this since the D3DX animation controller does it all for us.

```
void MyAnimController::AdvanceTime( double Elapsed )
{
    LPD3DXKEYFRAMEDANIMATIONSET pAnimSet = NULL;
    LPD3DXFRAME pFrame = NULL;

    D3DXVECTOR3 Scale, Translate;
    D3DXQUATERNION Rotate;
    D3DXMATRIX mtxFrame;
    ULONG i;
    char AnimName [1024];

    m_GlobalTime+=ElapsedTime;

   // Loop through each track
   for ( uint TrackIndex = 0; TrackIndex < GetNumTracks() ; TrackIndex++ )
```

```
   {
      // Increment track timer scaled by track speed
      m_Track[TrackIndex].TimerPositon+=ElapsedTime*m_Track[TrackIndex].Speed;

      // Get Anim set assigned to this track
      pAnimSet = GetAnimationSet( TrackIndex , &pAnim );

      // Map track time to animation set time
      float PeriodicPosition = pAnimSet->GetPeriodicPosition(
                                          m_Track[TrackIndex].TimerPosition );
      // Get the number of animations in this animation set
      ULONG NumAnimations = pAnimSet->GetNumAnimations();

      // Loop through each animation in this animation set and get the new SRT data
      for ( Animation = 0; Animation < NumAnimations; ++Animation )
      {

         // Get name of current animation
         // should match the name of the frame it animates
         pAnimSet->GetAnimationNameByIndex( Animation , &AnimName );

         // Generate the Scale, Rotation and Translation data for this animation
         // for the current periodic position
         // We looked at GetSRT earlier.
         pAnimSet->GetSRT( PeriodicPosition, Animation ,
                         &Scale, &Rotate, &Translate );

         // Generate a matrix from the SRT data.
         // This global D3DX function generates a 4x4
         // transformation matrix from a scale vector, a quaternion
         // and a translation vector. Check out the docs for details.
         D3DXMatrixTransformation( &mtxFrame, NULL, NULL, &Scale,
                                   NULL, &Rotate, &Translate );

         // Find the frame that this interpolator animates in the hierarchy
         pFrame = D3DXFrameFind(m_pFrameRoot, AnimName );

         // Update the frames matrix with out new matrix
         pFrame->TransformationMatrix = mtxFrame;

    } // Next Animation

    // Release the anim set we no longer need it
    pAnimSet->Release();

  } // Next Track
}
```

The first thing this function does is update the global time of the controller. This global timer may seem useless at the moment, but when we examine callbacks, you will see how the global timer becomes a much more useful feature. Next, we loop through each track and increment its timer. In this case we scale the elapsed time about to be added by the speed property of the track. We then convert the new track time into an animation set periodic position which we can feed into the GetSRT function. We then loop through each animation stored in the animation set and, for each one, call ID3DXAnimationSet::GetSRT to generate the new scale, rotation and translation information. We then

use the returned information to construct a matrix. Once the matrix is constructed, we search the hierarchy for a frame with the same name as the animation and replace its frame matrix with the newly generated one. The D3DX animation controller would not search the hierarchy in this way as it would be far too slow, which is why it stores the matrices in a separate array that can be accessed quickly. Indeed, this is why we register animation outputs in the way discussed earlier -- so the controller can know exactly which matrix is paired with which animation and quick matrix updates can be performed without the need for frame hierarchy traversal. However, as this function has been written with clarity of process in mind, we will forgive ourselves for bypassing this optimization.

One thing you may have noticed is that we are not multiplying the current frame matrix by the newly generated animation matrix. This is because all of the keyframe animation data stored inside an animation is ***absolute***. This is true of both the data in the .X file itself as well as in the data output by the SRT functions (by definition, because it is stored this way in the file). To be clear, what is meant by 'absolute' is that the keyframe data is not relative to what came before it (i.e. it is not incremental). Each keyframe is a complete snapshot that describes the transformation state of the object at that particular moment on the timeline. The transformations are still defined relative to the parent frame of course (it's still a parent-relative matrix), because we want the spatial hierarchy relationships to remain intact during animation. This means that we are able to overwrite the frame matrix without fear of having to preserve or take into account previous transformations. Each time we call AdvanceTime, the frame matrix will be repopulated with the next set of absolute SRT values. In other words, we get a new frame matrix for each animation update. To make the system work incrementally would be more challenging and would require that we track more data between update calls. Since this is typically unnecessary and does not add much value to the system, absolute values are preferred.

> **Note:** Although we are searching through the frames here (using D3DXFrameFind) to find the correct matrix in the hierarchy to modify, the animation controller does not need to do this. Recall that it has access to a pointer for each matrix in the hierarchy along with the name of the frame it is associated with. For more information you can refer back to our discussion of the ID3DXAnimationController::RegisterAnimationOutput function discussed earlier.

ResetTime

The final function we will cover in this section is called ResetTime. It allows the application to wrap the global time of the animation controller back to zero, much like we did in the earlier code with the track timers. This is especially handy if we have many animation sets that do not loop and would cease playing once the sum of elapsed time passed into the AdvanceTime method exceeded its period. This function causes the entire animation system to restart, taking smooth wrapping of track timers into account. This is especially useful when considering sequencing which we will discuss in the next section.

For now, just know that sequencer events (called key events) are registered on the global controller timeline (not the per-track timelines) and as such, once an event has been triggered when the global time has reached its timestamp, it will never be triggered again unless the global time is restarted. So we can imagine that a complex group of animation sets might wish to be looped at the global level so that the key events set on the global timeline are also re-triggered each time. If we simply configured each

animation set to loop, the animations themselves would loop but each key event on the global timeline would only be triggered once.

It is also worthy of note that one might also choose to reset the global time of the controller even if the sequencer is not being used, simply to stop the global time value reaching an extremely high value. Obviously this would only be a concern if an animation was left to play for a considerable period of time. The method is shown below.

```
HRESULT ResetTime(VOID);
```

The ResetTime method is actually a little more complicated than it might first seem. Not only does it reset the global timer of the controller back to zero, but it also wraps around each of the track timers. It does not simply set them to zero as this would cause jumps in the animation. Instead, it wraps them around such that the track timer is as close to zero as possible without altering the periodic position of the animation set currently being played.

During the call to ResetTime, the animation set's GetPeriodicPosition function is called, passing in the current track time. If the D3DXTRACK_DESC::Position was 137 prior to ResetTime(), then this is the position that gets passed to the GetPeriodicPosition call. Normally, GetPeriodicPosition returns the current time, after it has been mapped into the set's local timeline. In the case of looping for example, we can assume that this is a simple fmod() of the current track time with the period of the animation set as discussed earlier. During ResetTime, the information returned by GetPeriodicPosition is taken as a 'snapshot', and all internal time data is updated using this snapshot. The information returned by GetPeriodicPosition is used to update the track position. You can prove this by examining the track timer (its position) both before and after the ResetTime call. Meanwhile the controller’s global time is reset to 0.0.

To summarize, let us look at an example of what is happening here. Imagine we had an animation set with a period of 10 seconds and let us make the call to ResetTime after 35 seconds of application time have elapsed. Prior to the ResetTime call, both our track containing the animation set, and our global time contain the value 35.0 (assuming we maintained a constant track speed of 1.0). At this point we make our call to ResetTime. During the call, the animation set's GetPeriodicPosition function is called and it returns what is essentially fmod(TrackPosition, pAnimSet->Period) -- therefore a value of 5.0. Once the call has completed, we find that the controller’s global time has been reset to 0.0, and the track position has been merely wrapped back into a sensible range, and now contains a value of 5.0.

> **NOTE:** No other animation set function is called during the Reset procedure. In addition, there are implications with respect to the callback mechanism. We will discuss this later when examining the D3DX animation callback mechanism.

So we have now looked at how to advance our animations during runtime using a simple function call. We have also seen how we can retrieve the global time of the controller if we wish to allow the application the freedom to manipulate a certain track at key global times. However, while we could allow our application to control the track properties at certain global times using the method shown earlier, this is precisely what the animation sequencer is for, and is the subject of our next section.

10.9 The Animation Sequencer

Earlier in the chapter we mentioned that the animation controller contains a very useful event scheduler, referred to as the **sequencer**, which allows us to apply and adjust track properties at user defined global times. In this section we will discuss the animation sequencer; another key component of the D3DX animation controller. What we will see is that the global timer of the controller is more than just a counter for elapsed time passed into AdvanceTime calls; it is actually the heartbeat of a powerful tool.

Recall that the elapsed time that we pass to AdvanceTime increments both global and local timelines. The global time is maintained at the controller level and is the value that is returned in response to a GetTime function call. It is ultimately the amount of time that has passed in seconds since calls to AdvanceTime began (assuming that you have not called ID3DXAnimationController::ResetTime to reset the global time back to zero). Unlike the animation set's GetSRT function which uses track times, the sequencer works using the controller's global time value of the controller.

For example, let us imagine that we would like to schedule an event that disables an animation track at the global time of 12 seconds. Further, let us assume that the animation set we are working with is only 10 seconds in duration, but is configured to loop. In this example, the animation set would stop playing exactly 2 seconds into its second iteration. This event would trigger only occur once -- when a time of 12 seconds or greater is first reached on the global timeline. This same event will *not* be triggered again with further calls to AdvanceTime (unless ResetTime is called).

If we want an event to be periodic (i.e., triggered once every N seconds), then the application is responsible for providing the logic that wraps the global time before it calls AdvanceTime again. For example, using our 12 second (global time) trigger described above, if we were calling the following function repeatedly (passing in the time in seconds that has elapsed since the last frame), the event would initially be triggered 12 seconds into the animation loop. Notice however, that every 20 seconds we wrap the time back to zero. Thus, after the first time the event has been triggered, it is then re-triggered every 20 seconds thereafter (i.e. every time CurrentGlobalTime reaches 12 seconds again).

```
void AnimateHierarchy ( double TimeElapsed )
{
    CurrentGlobalTime = pAnimController->GetTime();

    if ( CurrentGlobalTime >= 20.0 ) pAnimController->ResetTime();

    pAnimController->AdvanceTime( TimeElapsed );
}
```

Just try to remember that events are scheduled in global time while animation sets use track time to loop independently. If your track speed is 1.0 and you have not altered the position of the track, then a 1:1 mapping will be maintained between global time and track time.

Let us look at another example of why this is useful. We will assume that we have a hierarchy that contains the mesh of a space station and the mesh of a space ship, and that we have defined two looping animation sets. The first animation set slowly rotates the space station frame along its local 'up' axis.

The second animation set is a 20 second animation set that animates the frame of the space ship so that it starts off docked inside the space station and then flies out of the space station and off into deep space (perhaps through a jump gate or wormhole). This sequence will make a space simulation more realistic as we come in to dock our ship, since we will see other space ships making use of the station as well. As you might imagine, simply looping the space ship animation set presents a bit of a problem. Seeing a new space ship emerge from the space station precisely every 20 seconds makes it quite obvious to the player that this is just a looping animation. We will generally prefer a less uniform and more random series of events as would be the case in the real world (using the term loosely of course given the circumstances). Making use of our sequencer provides an opportunity to improve our situation. We could very easily make the pattern a lot less obvious simply by choosing a more random distribution of global times to start and stop the animation (i.e. enable/disable the track). Imagine for example, that we set events for the second track on the mixer (the track our space ship animation is assigned to) to occur at the following global times (in seconds). Keep in mind that the space ship animation lasts for 20 seconds before looping.

Event Number	Global Time (GT)	Event Type
1	20	Track Disable
2	30	Track Enable
3	70	Track Disable
4	100	Track Enable
5	160	Track Disable
6	180	Track Enable

Our accompanying animation update function might look something like the following:

```
void AnimateSpaceSceneHierarchy(FLOAT TimeElapsed)
{
       if ( pAnimController->GetTime() > 200.0 )  pAnimController->Reset();
       pAnimController->AdvanceTime( TimeElapsed );
}
```

We see immediately that our sequencer loop is now on a much larger scale of 200 seconds with some event distribution in between. This should break up the pattern nicely over time and provide some additional realism. Here is how everything would play out…

The space ship track would start off enabled and the space ship animation would play out to completion once (it lasts for 20 seconds). Our first event occurs immediately (GT = 20 secs) which turns off the track, creating a 10 second gap where no animation is being played for the space ship. 10 seconds later (GT = 30 secs) our second event occurs which re-enables the space ship track. Note that during the 10 second time between the first two events, the animation controller will ignore the disabled track and will not increment the track's timer when AdvanceTime is called. Therefore, our space ship mesh would be frozen somewhere off in the distance (hopefully far enough away so as not to be visible!). Notice as well that we are very careful to disable the track on 20 second boundaries so that we do not disable the animation while the space ship is still in the middle of its sequence.

The track then plays for 40 more seconds causing the animation to play two more times (looping) before our next event occurs (GT = 70 secs). This third event disables the track again. Because the animation is

always being frozen on a 20 second boundary, we can be sure that whenever we enable it again it will start off at a track position close to zero where the ship is back inside the space station. Our fourth event (GT = 100 secs) re-enables the track and we let the track play for another 60 seconds. This means that the animation will loop three times, one immediately after another. We then disable the track again at event five (GT = 160 secs) and do not enable it again until our sixth event (GT = 180 secs) leaving a 20 second gap where no ships leave the space station. Finally our sixth and last event (GT = 180 secs) re-enables the track. Note that the looping would remain enabled and we would see ships take off every 20 seconds were it not for the logic in our AnimateSpaceSceneHierarchy function. Rather than continue to increment the global time, we roll it back to zero once we have had 200 seconds elapsed in total. Thus, the whole cycle would start again. Our final event which re-enables the space ship animation would actually play twice before being disabled again at the rolled around global time of 20 seconds.

Admittedly, this very simple example only scratches the surface of some of the creative ideas you can come up with for using the animation sequencer. But it is clear enough that just by registering a few track enable/disable events with the sequencer we have turned a 20 second looping animation into a 200 second animation sequence that is far less uniform and perceptible to the player. The gap between ships leaving the station would no longer seem fixed and it is unlikely that the player would pick up on this pattern. You could probably imagine experimenting with additional values like track speed as well so that the ships seem to fly faster or slower. Perhaps you might occasionally turn on blending with a second track that includes a different translation keyframe value so that ships fly off in different directions. Maybe you even would turn on an afterburner track at random intervals, etc.

With this high level overview in place, let us now examine how we can set key events on the global timeline using the controller's sequencing methods. We will also need to discuss the types of key events we can register.

10.9.1 Registering Sequencer Events

There are five different Event Key types that we can register with the animation sequencer. This is done via one of five corresponding methods exposed by the ID3DXAnimationController interface. So to set an event, we just call the function which matches the event type and we are done. It is very important that we do not try to register more events with the sequencer than the maximum number of events that the animation controller can process. Recall that we set this maximum number of events when we created/cloned the animation controller (the MaxNumEvents parameter).

> **Note:** D3DXLoadMeshHierarchyFromX creates an animation controller that can store a maximum of 30 events. Trying to register additional events beyond those 30 will fail. To increase the event pool size, we simply need to clone a new controller and specify the desired number of slots.

We can query the number of events the animation controller can have registered with it using the following method of the ID3DXAnimationController interface. This next function returns the maximum number of events that can be registered with the controller.

```
UINT GetMaxNumEvents(VOID);
```

Let us quickly discuss these five functions along with their parameters and the types of events they allow us to set. Notice how each of the event registration functions returns a D3DXEVENTHANDLE. We will discuss what these handles can be used for at the end of this section.

1. The KeyTrackEnable Function

This function allows us to schedule an event that will enable or disable a track on the animation mixer (a 'KeyTrackEnable' event). When a track on the mixer is disabled, any animation set assigned to that track is essentially frozen. It will not receive any updates from the AdvanceTime function and the track timer will not be incremented until it becomes re-enabled. Note that when we re-enable the track, it will not restart from the beginning -- it will resume from the time it was frozen. This allows us to freeze an animation at any point and then resume it whenever we wish.

```
D3DXEVENTHANDLE KeyTrackEnable(UINT Track, BOOL NewEnable,
                               double StartTime);
```

UINT Track
This parameter is the zero-based index of the mixer track we are registering the event for. For example, to disable the fifth track of the animation mixer (and its assigned animation set) at a specific global time, we would call this function with a value of 4 for this parameter.

BOOL NewEnable
This second parameter should be set to true or false depending on whether we wish to enable or disable the specified track at the specified time, respectively.

DOUBLE StartTime
The final parameter is the global time (in seconds) when we would like this event to occur.

Example:

```
pAnimController->KeyTrackEnable( 0 , FALSE , 35.0 );
pAnimController->KeyTrackEnable( 2 , TRUE , 100.0 );
```

The prior example registers two KeyTrackEnable events with the animation sequencer. In the first call to the function we specify that we would like the first track on the animation mixer (track 0) to be disabled 35 seconds into the animation (when the sum of elapsed time passed into the AdvanceTime function passes 35 seconds). In the second call to the function we register an event with the sequencer that specifies that at 100 seconds global time, we would like the third track of the animation mixer (track 2) enabled and to start playing its assigned animation set.

2. The KeyTrackPosition Function

The next type of event that we can register with the animation sequencer is referred to as a KeyTrackPosition event. This event allows us to set the timer of a track to a specific value (position)

when a specific global time is reached. In other words, we are saying that at a specific global time, we would like to instantly transition the track timer to a specific value regardless of the current global time. As it is the position of a track (the track timer) that is used to map to a periodic position, this function ultimately allows us to jump to a desired location in the animation set at a specified global time.

This can be a very useful event if you are using a single animation set that stores all of your animation sequences (ex. run, walk, jump, etc). Imagine that we had an animation set that contained only two animation sequences: idle and walk. You could quickly pre-script some AI-like events (or load them from a file) that basically has a character which idles for a few seconds and then starts walking, stops and idles again, walks a little more, etc. While we will do this differently in our demo engine (using separate animation sets and then enabling/disabling their tracks), if you are using single animation sets, this might be an approach to consider. Note that if you wanted your AI to make these decisions on the fly, rather than use a pre-programmed set of events, you would probably use the ID3DXAnimationController::SetTrackPosition to cause the timeline advancement to happen immediately. However, there are certainly cases where the event management concept can be put to good use. Your application could, for example, decide to register a track position change event to occur in 10 seconds time using the sequencing functions.

```
D3DXEVENTHANDLE KeyTrackPosition( UINT  Track, double NewPosition,
                                  double StartTime );
```

UINT Track
This is the index of the mixer track for which we would like this event to apply. It is the timer of this track which the event will manipulate.

double NewPosition
This parameter contains the new position that we would like this track's timer set to. For example, if this value was set to 200, then when this event is triggered, the track time would instantly be set to 200 and any future calls to AdvanceTime would add elapsed time relative to this. This advancement would happen at the global time specified in the StartTime parameter. Remember also that this is not the same as directly specifying a position in the animation set (although it can be). The track position is mapped to a periodic position so that looping is factored in, and it is the periodic position that is used to fetch the SRT data for each animation. Therefore, manipulating the timer of the track allows us to *indirectly* manipulate the periodic position generated for the animation set.

double StartTime
This is the global time (in seconds) when we would like this event to occur. When the global timer of the controller reaches or passes this value, the event will be triggered.

As another simple example, we might imagine that we have a zombie in our game that, five seconds after being hit by our bullet, disintegrates into a pile of dust and bones. Again, we are assuming that this disintegration sequence is part of a single animation set that might include other animation sequences like the zombie walking or attacking, etc. In the following example, the zombie animation set is assigned to track 0 on the animation mixer. When the zombie is hit by a bullet we register an event that advances the animation to the death sequence (position 200) in 5 seconds time (5 seconds being an arbitrary delay that we have chosen to delay the disintegration sequence).

```
if ( ZombieIsHitByBullet )
{
      double CurrentTime = pAnimController->GetTime ();
      pAnimController->KeyTrackPosition ( 0 , 200 , CurrentTime + 5.0 );
}
```

3. The KeyTrackSpeed Function

We mentioned earlier how the ID3DXAnimtionController::SetTrackSpeed function sets a scalar value (per track) to scale the amount that the track's local time is incremented using the passed global time. Setting this to 1.0 means that the animation set time is incremented normally. So we can see that each track has a clock and a speed value. We can set this speed value directly using the SetTrackSpeed function discussed earlier in the chapter or we can choose to let the sequencer do it at a certain global time using the KeyTrackAnimSpeed function.

```
D3DXEVENTHANDLE KeyTrackSpeed ( UINT Track, float NewSpeed,
                                double StartTime, double Duration,
                                DWORD Method );
```

DWORD Track
This is the index of the track we would like the event to be applied to.

float NewSpeed
This is the new speed value which we will set for this track. A value of 1.0 means the animation will play at its default speed (the speed intended by the animation creator). The speed value of each track is used to scale the amount by which the track's internal clock (i.e., the track position) is incremented. Setting this value to 3.0 will cause the animation controller to scale the elapsed time passed into the AdvanceTime method by 3, before adding it to the track timer.

double StartTime
As with all key event types that we can register with the sequencer, we always have to specify a global time at which this event will take place. If we set this value to 200, it means that this event will be triggered when the controller global timer reaches 200 seconds.

double Duration
Unlike the previous functions we discussed, this function allows us to specify an end time for our requested change so that we can transition gracefully between events. The `Duration` parameter specifies how long (in seconds) we want the transition between the old speed and the new speed to take. For example, assume that the current speed of the track is 10.0 and we set an event at 100 seconds global time to set the speed to 5.0. We also set the duration of the event to 10.0. Then starting at 100 seconds global time, the speed of the animation set assigned to this track would slowly decrease over the next 10 seconds until it finally reached the requested speed of 5.0. In other words, it would take 10 seconds to reach the requested speed of 5.0 -- we gradually ease in to it rather than abruptly change. The calculation used to scale between the current speed and the requested speed of the track is determined by the Method parameter discussed next.

DWORD Method
We use this parameter to pass a member of the `D3DXTRANSITIONTYPE` enumerator. It describes the method we would like to use to run the transition calculation between current and target values. There are two methods to choose from: `D3DXTRANSITION_LINEAR` provides a simple linear interpolation between the old speed and the new speed and `D3DXTRANSITION_EASEINEASEOUT` uses a spline-based curve to handle the transition. In other words, it is linear falloff versus exponential falloff (to refer back to some familiar material we discussed in Chapter Five).

Example

```
pAnimController->KeyTrackSpeed( 2 , 2.0 , 200.0 , 10.0 , D3DXTRANSLATION_LINEAR );
```

This example sets an event for the third mixer track (track 2) to be triggered at 200 seconds global time. It will begin to set the speed value for the track to 2.0, from whatever its current speed is, over a period of 10 seconds. Linear interpolation will be used to affect the changes in between.

4. The KeyTrackWeight Function

Every track on the animation mixer can be assigned a blend weight using the ID3DXAnimationController::SetTrackWeight function discussed earlier. This weight is used to scale the influence of the animation set on the respective frames in the hierarchy. We can also schedule a KeyTrackWeight event with the sequencer so that the weight of a track can be changed at scheduled global times.

Like the previous function discussed, we are able to apply our change over a specified duration. As one usage example, we might have an animation that violently shakes certain meshes in the hierarchy in response to a laser blast. We could schedule the weight of this 'shake' animation to decrease to zero over a period of N seconds, such that the effect of the blast on the hierarchy seems to dissipate with time (i.e. less and less noticeable shaking over time).

```
D3DXEVENTHANDLE KeyTrackWeight ( DWORD Track, float NewWeight,
                                 double StartTime, double Duration,
                                 DWORD Method );
```

DWORD Track
This is the track that we wish this event to be scheduled for.

float NewWeight
This is the new weight that we would like to set for this track. The default value is 1.0, so the animation affects the hierarchy at full strength. Setting this value to 0.5 for example would mean that the animation would affect the hierarchy at only half strength.

double StartTime
This is the global time when the event should occur.

double Duration
This is the span of time over which the transition from the previous weight to the new weight should take place. If we specified a new weight of 0.0 and the current weight was 1.0, and we used a Duration of 3 seconds, then once the specified global time was reached, the weight of the track would be slowly degraded from 1.0 to 0.0 over a period of 3 seconds.

DWORD Method
This parameter indicates the interpolation method that generates the intermediate values during the transition between current and target weight values. The two choices are linear interpolation (`D3DXTRANSITION_LINEAR`) or interpolation along a curve (`D3DXTRANSITION_EASEINEASEOUT`).

```
pAnimController->KeyTrackWeight(1, 0.0, 100.0, 3.0, D3DXTRANSITION_EASEINEASEOUT);
```

This example schedules a weight event to take place on track 1 at 100 seconds global time. The weight will be exponentially scaled down from its current weight to a value of 0.0 over a period of 3 seconds. Once done, the animation assigned to this track will not affect the hierarchy in any way.

5. The KeyPriorityBlend Function

The final event type we can register with the sequencer is the priority blending event. This is the sequencer replacement for the `SetPriorityBlend` function covered earlier. This function allows us to smoothly transition between weighting the blending between high and low priority track groups. The function is shown below with a list of its parameters. Notice that unlike the other sequencing functions we have discussed, there is no Track parameter. You should recall that this is because the priority blend is a property of the animation mixer itself, and not a specific track.

```
D3DXEVENTHANDLE KeyPriorityBlend( float BlendWeight,
                                  double StartTime, double Duration,
                                  DWORD Method );
```

float BlendWeight
This parameter is the new priority blend value for the animation mixer. It should be a value between 0.0 and 1.0. This value can be thought of as setting the strength of low priority tracks in the final blending operation. A value of 1.0 would mean that the high priority tracks do not influence the hierarchy at all.

double StartTime
The global time in seconds at which the event should be triggered.

double Duration
The length of time it should take to transition between the current and target priority blend weights.

DWORD Method
This parameter indicates the interpolative method that generates the intermediate values during the transition between current and target priority blend weight values. The two choices are linear interpolation (`D3DXTRANSITION_LINEAR`) or interpolation along a curve (`D3DXTRANSITION_EASEINEASEOUT`).

10.9.2 Event Handles and Working with Registered Events

We have now seen how easy it is to register an event with the sequencer; a simple function call and the work is done. However, there may be times when you wish to reset the global time of the controller back to zero, but do not wish a certain event to occur a second time through. In order to address this situation, we need a function that allows us to remove an event from the global timeline as well as register one. The ID3DXAnimationController provides us such a method. It also provides methods for querying upcoming events, query current events being executed, and validating events as still active or not.

In order to ask the animation controller for details regarding an event, we need some way to inform it which event we are enquiring about. You will likely have noticed that each of the functions that register a key event returns a D3DXEVENTHANDLE. This handle can be stored and used to enquire about that event at a later time.

`HRESULT UnkeyEvent( D3DXEVENTHANDLE hEvent );`

This function allows the application to remove a registered event from the sequencer. You simply pass in the handle of the event. Below we see an example of a TrackEnable event that has been registered to occur at 100 seconds global time. We then use the returned handle to later remove it from the timeline. Ultimately, this allows us to discard events which we no longer need, which can be useful if your animation controller has been configured to store only a small number of events.

```
D3DXEVENTHANDLE Handle;
Handle = pAnimController->KeyTrackEnable( 2 , TRUE , 100.0 );
..
..
..
pAnimController->UnkeyEvent( Handle );
```

As you can see, the handle returned from the KeyTrackEnable function is used to identify the event to the controller in the UnkeyEvent method.

`HRESULT UnkeyAllTrackEvents( UINT Track );`

This function does not require an event handle to be passed in its parameter list because it is used to remove *all* events scheduled for a given track. This will not affect any events that have been registered for other tracks. Of course, it cannot be used to unregister priority blend events, since priority blend events are global to the controller and are not assigned to any track. There is a separate function (discussed next) for removing all events of this type.

In the following example we register three events with the controller. Two of these events are registered for track 2 on the mixer at 100 and 300 seconds respectively, and an event is also registered for track 3 to occur at a time of 200 seconds.

```
pAnimController->KeyTrackEnable( 2 , TRUE , 100.0 );
pAnimController->KeyTrackEnable( 3 , FALSE ,200.0 );
pAnimController->KeyTrackEnable( 2 , TRUE , 300.0 );
…
…
pAnimController->UnKeyAllTrackEvents( 2 );
```

Notice that after the three events have been registered, later in the code we unregister all events assigned to track 2. In the above example, this would remove only two of the three events. The event that was registered for track 3 at a global time of 200 seconds would still remain on the global timeline.

HRESULT UnkeyAllPriorityBlends(VOID);

When priority blend events are scheduled using the KeyPriorityBlend function, these events alter a global setting of the controller, and as such are not specific to any track. Therefore, if we wish to remove all priority blend events that have been registered with the animation sequencer, we must use the UnkeyAllPriorityBlends function. This function will remove *all* events that have been previously registered with the KeyPriorityBlend function.

HRESULT ValidateEvent(D3DXEVENTHANDLE hEvent);

The ValidateEvent function can be used to test if an event is still valid. Valid events are events which either have not been executed yet (because the global time is less than the event timestamp) or events that are currently being executed.

In this next example we show an event that has been registered for track 2 at 100 seconds global time. Elsewhere in the code, the event is validated. If the event is found to be invalid then it means it has already been triggered and has completed. In this case, we then remove the invalid event from the timeline.

```
D3DXEVENTHANDLE Handle;
Handle = pAnimController->KeyTrackEnable( 2 , TRUE , 100.0 );
..
..
..
if ( FAILED ( pAnimController->ValidateEvent( Handle )) )
     UnKeyEvent ( Handle );
```

This method provides a way for our application to know whether a certain event has been triggered yet. However, you should be careful if you intend to reset the global time and wish the events to be triggered again each time through the global timeline. Once the event is gone, it is gone for good. If the application was then to call the ResetTime method resetting the global time of the controller to zero, the same event would not be triggered when the global time reaches 100 seconds (in the above example) a second time because it has been removed from the timeline

```
D3DXEVENTHANDLE GetCurrentTrackEvent( UINT Track,
                                      D3DXEVENT_TYPE EventType );
```

This function allows you to query the current event that is running on a particular track. This allows the application to fetch the event handle that is currently being executed. This event handle can then be passed into the GetEventDesc method (discussed below) to fetch the event details.

For the first parameter, the function is passed the index of the track for which we wish to fetch the handle of the event currently being executed. The second parameter allows us to specify a filter so that we can fetch the event only if it is of a certain type. If no event of the specified type is currently running on the specified track, the function will return NULL.
For the second parameter, we should pass in a member of the D3DXEVENT_TYPE enumeration:

```
typedef enum _D3DXEVENT_TYPE {
    D3DXEVENT_TRACKSPEED = 0,
    D3DXEVENT_TRACKWEIGHT = 1,
    D3DXEVENT_TRACKPOSITION = 2,
    D3DXEVENT_TRACKENABLE = 3,
    D3DXEVENT_PRIORITYBLEND = 4,
    D3DXEDT_FORCE_DWORD = 0x7fffffff
} D3DXEVENT_TYPE;
```

Passing in D3DXEVENT_PRIORITYBLEND to this function will always return NULL because a priority blend event is not a track event. The current code shows how we might query for the currently running track speed event (if any) on track 2 and output the speed value. This could be useful for debugging or perhaps for slowing down in-game MIDI music based on the speed of a track.

```
D3DXEVENTHANDLE Handle;
Handle = pController->GetCurrentTrackEvent( 2 , D3DXEVENT_TRACKSPEED );

if ( Handle )
{
     D3DXEVENT_DESC desc;
     PController->GetEventDesc( Handle , &desc );
     Double Speed = desc.Speed;

     if ( Speed < 0.5 ) PlaySlowMusic();
     else
                        PlayFastMusic();
}
```

In the above pseudo-code, the handle of any currently playing speed change event is requested. If a speed event is currently being triggered on track 2, we are returned the handle (or NULL if no speed event is currently being carried out).

If an event is being performed, then we use the GetEventDesc method of the controller (discussed shortly) to fetch the details of the event. They are returned in the passed D3DXEVENT_DESC structure. In this example, the speed of the event is queried and used to play different music depending on the speed at which the animation is playing.

```
D3DXEVENTHANDLE GetCurrentPriorityBlend(VOID);
```

This method is the sister function of the function previously described. As discussed previously, priority blend events are different from all other key events in that they are not assigned to a specific track. As such, a different function is exposed to allow the application to retrieve the event handle of any priority blend event that is currently being executed.

The function takes no parameters and returns the event handle of the current priority blend event being executed by the controller (or NULL if no priority blend event is currently in progress).
Once you have the event handle, you can then call the GetEventDesc method (just as we did in the prior code example) to fetch the details of the priority blend event. This will tell us things like its time stamp, its start time, weight, and duration.

```
HRESULT GetEventDesc( D3DXEVENTHANDLE hEvent,  LPD3DXEVENT_DESC pDesc );
```

This function is useful for extracting the information for any event for which the application has an event handle. By simply passing in the event handle as the first parameter and a pointer to a D3DXEVENT_DESC structure, on function return, the details of the event will be stored in the passed structure.

We used this function in the description of the GetCurrentTrackEvent example above, but it can also be used to retrieve the information of a priority blend event (if the handle to a priority blend event is passed in as the first parameter).

The D3DXEVENT_DESC will contain all the information about the event:

```
typedef struct D3DXEVENT_DESC {

    D3DXEVENT_TYPE      Type;
    UINT                Track;
    DOUBLE              StartTime;
    DOUBLE              Duration;
    D3DXTRANSITION_TYPE Transition;

    union
    {
        FLOAT  Weight;
        FLOAT  Speed;
        DOUBLE Position;
        BOOL   Enable;
    }

} D3DXEVENT_DESC, *LPD3DXEVENT
```

As we might expect, the first five members describe the details of the event itself (e.g. what type of event it is -- D3DXEVENT_TRACKSPEED or D3DXEVENT_PRIORITYBLEND). We are also returned the track number that the event applies to (not applicable to priority blend events) and the global time for the event trigger and the duration (in seconds) that the event will take to fully execute. We can also examine the transition type that is being used for this event to reach its desired final state.

We discussed the various events that can be scheduled earlier, and saw that duration and transition information is not applicable to KeyTrackEnable events or KeyTrackPosition events.

Finally, the last member of this structure is a union. The member that is active in this union depends on the type of event that it is. For example, if the Type parameter is set to D3DXEVENT_TRACKWEIGHT or D3DXEVENT_PRIORITYBLEND then the Weight member of this union will contain the Weight that this event will set the track to use or set the priority blend to. If the event type is D3DXEVENT_TRACKENABLE then the Enable Boolean will be set to true or false depending on whether this event is configured to enable or disable the current track.

```
D3DXEVENTHANDLE GetUpcomingTrackEvent( UINT Track,
                                       D3DXEVENTHANDLE hEvent );
```

This function allows us to pass in an event handle to get back the next event that is due to be executed from the current event (not including the event passed in) for the specified track. This function actually searches the global timeline from the event passed in to find the next event that is scheduled. When an event is found, not only is its handle returned, but it is also cached in the controller. While this does not alter the performance or playback of events, it does allow us to subsequently call this function repeatedly, passing NULL as the event handle. Each call will step to the next event in the event queue for the specified track, until the function returns NULL. NULL indicates that there are no more events for this track. This allows us to iterate through all future events until the end of the event queue is reached.

If we pass in NULL to the initial call to this function, the handle to the next scheduled event (from the current global time) will be returned and cached for future calls.

```
D3DXEVENTHANDLE GetUpcomingPriorityBlend( D3DXEVENTHANDLE hEvent );
```

This function is the priority blend event version of the previous function. We pass in the event handle of priority blend event (or NULL to start searching from the current global time) and it will return the handle of the next scheduled priority blend event.

This function (like the previous) can be called iteratively to step through all future priority blend events that may exist. When the function returns NULL, there are no further priority blend events scheduled on the timeline and we have reached the end of the event queue.

10.9.3 Animation Sequencer Final Notes

The sequencer does not attempt to optimize or compact its event set, so do not make any assumptions about its behavior. For example, each time you pass in a value with the same global time, even if the events are identical in every way, a new event is added to the internal queue. As you can imagine, we can very quickly hit our `MaxNumEvents` limit defined for the animation controller if we are not careful. The following code would actually register 5 events with the sequencer even though they are exactly the same.

```
m_pAnimController->KeyTrackEnable ( 0 , FALSE , 5 );
m_pAnimController->KeyTrackEnable ( 0 , FALSE , 5 );
m_pAnimController->KeyTrackEnable ( 0 , FALSE , 5 );
m_pAnimController->KeyTrackEnable ( 0 , FALSE , 5 );
m_pAnimController->KeyTrackEnable ( 0 , FALSE , 5 );
```

While we would probably not do something as obvious as the above example, we do have to be very careful that we do not register events inside frequently called snippets of code. For example, a careless programmer might intend to register a single event with the animation controller, but place the code that registers it inside his main animation or render loop:

```
void UpdateAnimations ( double Time )
{
      pAnimationController->KeyTrackEnable( 0 , FALSE , 5 );
      pAnimationController->AdvanceTime ( Time );
}
```

The above code does more than just waste cycles. Every time this function is called a new event will be registered that is the same as all of the others. Our available event slots would fill up very quickly (potentially within a fraction of a second) and once the animation controller event list is full, any future calls to register additional events will fail. Therefore, you have to make sure that you set events only where and when you need them and make sure that you only set them once.

10.10 The D3DX Animation Callback Mechanism

In the DirectX 9 summer update of 2003, Microsoft added a very useful callback mechanism to their animation system. This feature allows our application to register callback events within an animation set by registering one or more callback keys. Callback keys are similar to SRT keys in that they each have a timestamp, which is specified in ticks within the period of the animation. Unlike SRT keys which are registered per-animation however, callback keys are global to all animations in the set. When the animation set is first created, these callback keys must be allocated and passed into the creation function. With each callback key, we also have the ability to pass in a pointer to a context (i.e., a pointer to any application specific data we require). When the callback event is triggered as the periodic position of the set reaches the callback key's timestamp, this context data is forwarded to an application defined callback function.

> **Note:** We have mentioned previously that the period of an animation set can be thought of as the S, R, or T keyframe within all animations with then highest timestamp (converted to seconds). As callback keys are also defined in ticks, and make up the fourth key type, they too are defined within the period of the animation set. Therefore, if we have an animation set with one animation whose last keyframe has a timestamp of 20 seconds, we would say that the period of the animation is 20 seconds. However, if we also registered a callback key at 25 seconds, the period of the animation would now be 25 seconds. It is very important that you think of callback keys as no different than SRT keys in this regard. Thus, in reality it is the scale, rotation, translation, and callback keys define the period of the animation.

An obvious case where this callback mechanism might be useful is the synchronization of sound effects with your animations. You may wish your application to play a sound effect when a certain point in the animation is reached, perhaps an explosion, for example. Using this system, your application could register a callback key with the animation set in which the explosion sound is triggered when the explosion animation sequence starts. In this case, the callback time stamp would be equal to the time (in ticks) that the animation set's explosion animation would begin. When the key is registered, we could also register some user defined data, perhaps a string containing the name of the .wav file to be played. When the event is triggered by the controller, a callback function would be executed and would be passed the name of this .wav file. Your callback function could use the passed name to actually play the sound.

So in many ways, callback keys are similar to the event keys discussed in the previous section. The main difference is the timeline used by each. As we have learned, event keys are registered on the global timeline while callback keys are registered within the local timeline of the animation set.

In our earlier discussion of animation sets, we skipped over parameters and functions that dealt with callback keys. Now we will revisit the animation set discussion and review the callback mechanism. It is important to understand exactly how the animation set deals with the callback keys behind the scenes, especially if we intend to write our own custom animation sets in the future. Therefore, we will discuss how callback keys are stored and registered with an animation set and how the animation controller gets access to that key data in order to execute the callback function.

10.10.1 Allocating Callback Keys in the Animation Set

Earlier we discussed the global function D3DXCreateKeyFramedAnimationSet, which an application can use to manually create a keyframed set of animations. Recall that we skipped the discussion of two parameters to this function; they are highlighted in bold below.

```
HRESULT WINAPI D3DXCreateKeyframedAnimationSet
(
        LPCSTR pName,
        DOUBLE TicksPerSecond,
        D3DXPLAYBACK_TYPE Playback,
        UINT NumAnimations,
        UINT NumCallbackKeys,
        CONST LPD3DXKEY_CALLBACK *pCallKeys,
        LPD3DXKEYFRAMEDANIMATIONSET *ppAnimationSet
);
```

As you may have already gathered from examining the function definition, it is at animation set creation time that we specify the maximum number of callback keys we intend to register. This allows the callback key array to be allocated when the animation set is created. Thus we see that the callback keys belong to the animation set and not to any particular animation in that set.

As the fifth parameter, we pass in the maximum number of callback keys our application might want to register with this animation set. With the sixth parameter, we are given a chance to populate this array with callback keys. While there is a separate function of the ID3DXKeyframedAnimationSet interface that allows us to store the key values in this array after the animation set has been created, often we will know the callback keys we wish to register when the set is created. So we can pass in a pointer to this application allocated callback key array at animation set creation time using this parameter. When the animation set is created, a callback key array of NumCallbackKeys will be allocated. If the sixth parameter is not NULL then this many keys will also be expected in the array passed in. The callback key data passed in is copied into the animation set's internal callback key array, so on function return, the array of D3DXKEY_CALLBACK structures can be deleted.

We now know that we allocate the memory for the callback keys at animation set creation time and can optionally pass in the callback keys at this time as well. So let us have a look at the D3DXKEY_CALLBACK structure to determine what needs to be passed into the D3DXCreateKeyFramedAnimationSet function. Keep in mind that each element in the input array represents exactly one callback key. Therefore, if we wished to register five callback keys, we would pass in a pointer to an array of five D3DXKEY_CALLBACK structures.

```
typedef struct _D3DXKEY_CALLBACK
{
    FLOAT Time;
    LPVOID pCallbackData;
} D3DXKEY_CALLBACK, *LPD3DXKEY_CALLBACK;
```

From the animation system's perspective, the only significant thing about the key is the first member of this structure. This is the timestamp of the event when we want this event to trigger a callback function. The important point to remember is that the timestamp should be specified in ticks (i.e., a periodic position) just like keyframes. As we know, this can be different from global time, especially if looping is enabled for this animation set. For example, if a looping animation set had a period of 10 seconds and we had callback key registered at 5 seconds, the callback key would be triggered at 5 seconds, and then every 10 seconds thereafter (i.e., track positions 5, 15, 25, 35, …). Once again, this is because the track time is mapped to a periodic position which is used to fetch SRT *and* callback keys that need to be executed.

> **Note**: Callback key timestamps are specified in ticks, just like keyframes. The animation set's 'TicksPerSecond' handles the conversion into seconds when needed. Callback keys are defined within the period of the animation set, just like SRT keyframes. This is important because if callback keys were registered on the global timeline (like event keys), then the callback key would not be triggered on looping animations. In our sound effect example, the sound would only play the first time though and never play again unless we manually reset the global time of the controller. Therefore, as callback events are used to allow the application to synchronize external events with animations being played, it is only logical for the callback keys and the animation data to be defined within the same timeline.

The second member of the D3DXKEY_CALLBACK structure is the context data, which will be passed to the callback function when the key is triggered.. This void pointer can be NULL or it can point to any application defined data/structure you desire. For example, let us imagine that we wanted to create an animation set which fired a callback at three different times throughout the animation. Assume the goal is to play sound effects: the first callback is triggered to play an explosion, the second is triggered to play a splashing water sound effect, and a third is triggered to play a door opening sound. We might set up the three callback keys and create the animation set like so:

```
// Somewhere in code the names of the sound effects are stored
TCHAR Sounds[3][128];
_tcscpy( Sounds[0] , "Explosion.wav" );
_tcscpy( Sounds[1] , "Splash.wav" );
_tcscpy( Sounds[2] , "DoorOpen.wav" );
..
..
..
D3DXKEY_CALLBACK keys[3];

//1st  Callback fired at a periodic position of 10 seconds
Keys[0].Time = 10.0f;
Keys[0].pCallbackData = (LPVOID)Sounds[0];

// 2nd Callback fired at a periodic position of 20 seconds
Keys[0].Time = 20.0f;
Keys[0].pCallbackData = (LPVOID)Sounds[1];

// 3rd Callback fired at a periodic position of 30 seconds
Keys[0].Time = 30.0f;
Keys[0].pCallbackData = (LPVOID)Sounds[2];

//Create an animation set called "MyAnim" which has 100 animations and has
// keyframes specified at
// 50 ticks per second. Also the set has three call-backs keys.
```

```
ID3DXKeyframedAnimationSet  *  pAnimSet = NULL;
D3DXCreateKeyFramedAnimationSet("MyAnim",50,D3DXPLAY_LOOP,100 ,3, &keys, &pAnim");
```

Two things about the previous code may seem odd when we consider how event keys work (see previous section). First, at no point do we specify the track number for which these events are to be assigned. Of course, that is because there is no need; callback keys are part of the animation set itself. Second, we are not specifying the callback *function* that is to be called when these callback events occur. As we will see in a moment, a pointer to this callback function, or to a class that contains this callback function, is actually passed into the AdvanceTime method (perhaps you remember that second parameter?). When AdvanceTime is informed by the animation set that a callback event has happened, it will retrieve the context pointer of the callback key (in our example this was a string containing the name of a sound effect) and pass it to the application-defined callback function pointer passed into AdvanceTime.

One advantage of passing the callback pointer to AdvanceTime is that we can easily change the callback function as game conditions change. The downside is that all callback keys that are triggered during a single AdvanceTime call (on all tracks) must use the same callback function. This unfortunately encourages the development of callback functions that degenerate into huge switch statements that cater for all events that may have been triggered. It might have been nice to allow the application to register the callback function pointer with the callback key itself, just like we did with the context pointer. That way, each callback could have had its own callback handler function, leading to a cleaner implementation. However, this small disadvantage is something we can certainly put up with given the overall benefits of having a callback system at all.

10.10.2 Callback Key Communication

Earlier we looked at the communication pipeline between the animation controller and its animation sets. As long as the animation set implemented the eight functions of the base class ID3DXAnimationSet, the animation controller has all it needs. The controller calls the methods of the animation set base class to perform tasks such as map the track time to a period position, fetch the SRT data for that period position, fetch an animation by index or name or fetch the entire duration/period of that animation. When we covered the ID3DXAnimationSet interface (the base class) earlier in the chapter we examined all of these methods and how they work, except one.

The one method of the base interface which must be implemented by all animation sets is called GetCallback. It is responsible for returning a callback key back to the animation controller. The method is shown next along with a list of its parameters. We will explain how this function works in some detail, because if you intend to implement your own custom animation sets, you will need to know how to implement a function of this type. When discussing this function, we will discuss the way that ID3DXKeyframedAnimationSet implements it.

> **NOTE:** How this method works may seem a little confusing at first. It will start to make more sense when we examine the way the animation controller predicts and caches the time of the next callback key. This will prevent it from having to call ID3DXAnimationSet::GetCallback for each of its animation sets in every AdvanceTime call.

The GetCallback method is a callback key querying function with a few clever tricks up its sleeve. The animation controller passes in the time where it would like the search to start and the method will return the next callback key in the animation set's timeline that is scheduled for execution in the future. What is interesting is that the animation controller passes in the search start time in track time and it expects to get back the next scheduled event in track time as well. So the GetCallback function has to take the passed track time, map it to a periodic position in the set, and then use the periodic position to start the search for the next callback key. Once the key is found, its timestamp is converted into seconds and is added to the initial track position. This is important to note. The function returns (to the animation controller) the time of the next scheduled callback event as a track position, not as a periodic position. This track position is cached by the controller and when it is reached, the callback is triggered.

For example, let us imagine that the first time AdvanceTime is called, it calls the GetCallback method of its animation set (we will assume only one track is active for now) and passes in a position of 0 seconds (i.e., the initial track position). The GetCallback method would start its key search from 0 seconds and will return the first callback key it finds after that time. If we imagine that a key was registered for 100 seconds, then the function would return a CallbackPosition of 100 along with the context data pointer that was registered with that key.

At this point the controller knows that the next callback event will not happen for another 100 seconds. Therefore, it does not have to call GetCallback for this track again until the 100 second mark has been reached. All it has to do in future calls to AdvanceTime is test the current track position against the cached next callback key time of 100 seconds. Only when the track position reaches or exceeds the currently cached callback time does work have to be done. At the 100 seconds mark, the callback function which is passed into the AdvanceTime method is executed with the callback key context (also returned from GetCallback and cached along with the key track time) passed as input. In our earlier example, this was the sound effect filename. Once the callback function completes execution and program flow returns back to AdvanceTime, the function will ask the animation set (via GetCallback) for the next callback key. If one exists, it will be cached just like the previous key, and the process starts over again.

Before we look at this procedure in more detail, let us take a look at the ID3DXAnimationSet::GetCallback function and talk about how it must be called by the animation controller. Remember, the purpose of this function is to take a time value as its parameter and return the next scheduled callback key, so that the animation controller can cache it.

```
HRESULT ID3DXAnimationSet::GetCallback
(
    DOUBLE Position,
    DWORD Flags,
    DOUBLE *pCallbackPosition,
    LPVOID *ppCallbackData
);
```

DOUBLE Position

This parameter will contain the initial track time (in seconds) from which we would like the next callback key returned. Usually, when the AdvanceTime method calls this function, it will not actually

pass in the current track time but will instead pass in the previously cached callback track time. If this was not the case, we would run the risk of missing callback events between calls.

For example, assume that we have a non-looping animation with callback keys registered at 10 and 12 seconds. Initially (when AdvanceTime is first called) the key at 10 seconds would be cached by the controller. Now imagine that we were running at a very slow frame rate and the next time AdvanceTime was called, 13 seconds had passed. The AdvanceTime method would detect that the current track position (13 seconds) is higher than the currently cached next event time (10 seconds) and would trigger the 10 second callback. This is exactly how it should behave up to this point. Now that the callback has been handled, the controller will now try to cache the position of the next scheduled event. However, if the current track position of 13 seconds was passed into GetCallback, we would be incorrectly informed that no callback keys remain. But this is not true as we know there is one at 12 seconds. Unfortunately, because we began our search for the next event at 13 seconds, we skipped it. So to avoid this issue, instead of passing in the actual track position to GetCallback, we pass the current cached time for the callback event we just handled (10 seconds in our case). Therefore, even though we are at 13 seconds actual track position, we begin the search for the next event at 10 seconds. This returns the next scheduled event at 12 seconds which is cached by the controller and executed in the next call to AdvanceTime.

> **Note:** Recall that callback key timestamps are specified in ticks. GetCallback will convert the passed value (in seconds) into a periodic position in ticks using the animation set's TicksPerSecond value. This allows it to locate the correct next scheduled callback key. Once the key is found, the timestamp is converted back into seconds and added to the start position (track time). This gives us a timestamp in track time instead of a time within the period of the animation set. It is this track position which is returned and then cached by the controller.

The controller does not try to play 'catch up' in the AdvanceTime function; it will only trigger one event (per track) within a given call. So at the end of the AdvanceTime call in our example, we have a current track position of 13 seconds and a cached next event time of 12 seconds. If another 13 seconds passes before AdvanceTime is called again, this would put the actual track position at 26 seconds. The AdvanceTime method would correctly detect that the current track position (26 seconds) is greater than the cached key time (12 seconds) and would execute the callback (admittedly, 14 seconds late). It would then pass the cached position (12 seconds) into the GetCallback method to begin a new search for the next scheduled event. As there were only two events in this example and looping is disabled, no further event is returned and cached. If looping were enabled, then the search would loop around and the next callback key returned would be the first one at 10 seconds. However, it would correctly detect that a loop had been performed and take this into account when returning the time of the callback as a track position.

Initially, this function can be confusing because it accepts and returns a track time, but works internally with data defined as periodic positions (just like the GetSRT method). So why does this function not simply accept a periodic position rather than a track position? The answer is very simple; if this function is passed a periodic position, it would have no idea how many loops of the animation have happened before the initial search time. Therefore, even when the callback key is found (which we remember is defined in local time), the function has no way of mapping it to a valid track position. If we imagine that a looping animation set with a period of 10 seconds has its GetCallback method called with a track position of 55 seconds, the function can correctly calculate that the animation has looped 5 times (50

seconds) and is currently in the fifth second of the fifth loop. This is very important because the function will need to be able to determine a proper offset within the period. If the animation set had a callback key at 8 seconds, we would find this key next (8 seconds) and would add it to the number of seconds this animation has been played out *in full* (50 seconds) to give as a track position of 58 seconds. Notice that we are talking about *completed* animations when we deal with looping. In this instance for example, we know that the event happens on the eighth second of a ten second animation. That should always be consistent, so we must offset from the start of a new looping cycle when we calculate the track time to trigger the event. This value would then be returned to the controller and cached. Obviously, if this function was not passed a track position and received only a periodic position, in the above example, a value of 5 (instead of 55) would have been passed. While we would have successfully located the next callback key at 8 seconds local time, we would have no way of knowing how to map this to track time. For further review, there is a much more detailed discussion of how these individual values are calculated and used, along with source code examples, in the workbook accompanying this chapter.

As discussed earlier, in nearly all cases, the actual track position is not used by the callback mechanism to perform the search for callback keys. Rather, we are jumping from one cached time to the next. This means that in extreme cases, the actual position we are passing into the GetCallback function may be quite different from the actual track position. As AdvanceTime is called many times per second and callback events are typically not set very close together for a single track, the animation system can catch up quickly even if it lags slightly behind the track position. This is true regardless of whether the time update has skipped past the times of many keys. For example, even though only a single callback is dealt with in a single call to AdvanceTime, typically AdvanceTime will be called somewhere between 20 to 60 times per second. This provides adequate time to quickly process any queued events and get back in synch.

While the animation controller usually passes the currently cached callback key time into the Position parameter of the GetCallback function, this is not always the case. The exception to the rule happens when the controller is in its callback key setup phase. In that case, there is no currently cached time or data. Indeed the cache is going to be invalid at several times throughout the controller's life. For example, the first time AdvanceTime is called there is no previously cached callback time. When this is the case, the actual current track position is passed in and the search for the next callback key is done from there. The cache is also invalidated if the controller time is reset to zero using ResetTime or if the track position is altered using SetTrackPosition. In all of these cases, the cache is considered dirty and the next time GetCallback is called by AdvanceTime to cache the next scheduled key time, the actual track position is passed in. Just to make sure this is clear and we understand why this must be the case, imagine that we had a currently cached time in the controller of 100 seconds, but then our application manually set the track position back to a time of 10 seconds. If we did not flush the cache, then the next time AdvanceTime is called, it would not trigger a callback until a track time of 100 seconds (the previously cached time before the reset) was reached. However, there maybe callback keys at 11 and 20 seconds that would be ignored. So in this example, when the track position was reset to 10 seconds, the actual track position would be passed into GetCallback the next time round. This would return the next scheduled event at 11 seconds, which is then cached by the controller. For all further calls to AdvanceTime, the previously cached time would be used again as the base position of the search.

> **NOTE:** The *actual* track position used to perform the callback key search only when the cache is invalidated via manipulation of the global or track timers.

DWORD Flags

The second parameter to the GetCallback function is also vitally important. If you intend to derive your own custom animation set from ID3DXAnimationSet, then you will need to understand these flags and perform your callback key search correctly, when they are passed in by the controller. This value can be zero, one, or a combination of the following flags defined in the D3DXCALLBACK_SEARCH_FLAGS enumerated type:

```
typedef enum _D3DXCALLBACK_SEARCH_FLAGS {
    D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION = 1,
    D3DXCALLBACK_SEARCH_BEHIND_INITIAL_POSITION = 2,
    D3DXEDT_FORCE_DWORD = 0x7fffffff
} D3DXCALLBACK_SEARCH_FLAGS;
```

D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION

This flag indicates that the controller would like the animation set to search for a callback key starting at the passed time, but not to include any key that is scheduled at that time. If you develop your own animation set, you will find that the controller will always pass this flag, except when the cache has been invalidated. The reason is fairly obvious when we consider what we already know about the normal operation of the callback cache.

For example, imagine that the previously cached callback key time was 10 seconds and that it has just been passed and processed. At this point the controller wishes to search for the next callback key, starting at 10 seconds. Of course, we know that there is already a callback key at 10 seconds since it is the one we just executed. Since the search function would logically find this key first, we would wind up getting stuck processing the same event over and over again. So by passing the D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION flag, the controller is instructing the animation set to search for a callback key starting at 10 seconds but to ignore the key at 10 seconds.

This completely solves any infinite looping problem and raises the question of why this flag is needed at all. After all, why not just make this behavior the default for the GetCallback function all the time? We always wish to exclude the initial position, do we not? Actually, no. Consider what happens when the cache has been invalidated because either the global or track timer has been manually altered or that this is the first time AdvanceTime has ever been called. You will recall that when this is the case, we actually perform the search starting from the real track position. If we have no previous information in the cache, we absolutely do want to search for the next callback key and include the current position. Imagine that we have a callback key at time 0.0 seconds and that at 10 seconds of track time we reset the position of the track to zero (thus invalidating the cache). At this point, the GetCallback function would be called and the track position passed would be 0.0 seconds. In this scenario, if we passed this flag, then the callback registered at 0.0 seconds would be incorrectly skipped. Therefore, we recognize that when the cache has been invalidated and the real track time is used to perform the next key search, this flag should not be specified. For all other times, it is.

D3DXCALLBACK_SEARCH_BEHIND_INITIAL_POSITION
This flag is used if you wish to search backwards from the currently passed position to find a previous callback key in the animation set's callback array. The controller itself will not typically pass this flag to the GetCallback method, but it can be useful for an application to use to retrieve a previously executed callback. If you have implemented your own animation controller, this feature might be very useful for playing backwards animations.

DOUBLE *pCallbackPosition
This parameter is where the controller passes in the address of a double which, on function return, should contain the timestamp of the next scheduled callback event in track time (seconds). The controller will cache this time so that it knows that no work has to be done and no callbacks have to be executed until this time has been reached. The conversion of the next scheduled key's timestamp from ticks into seconds is performed by the function before the value is returned in this parameter.

LPVOID *ppCallbackData
The controller uses this output parameter to get a pointer to the context data that was registered for the next scheduled key. In our earlier example, each key stored a sound filename in its context data area, thus it is the pointer to this string that would be returned here. The animation controller caches this callback data pointer along with the cached time of the event. When the cached event time is reached, the callback handler is called and the cached callback data (i.e., context data) is passed to the handler for processing. In our example, the callback function would be passed the string name of the sound effect and it could then play that sound.

While the callback system may seem complicated at first, it is actually quite simple. The following few lines of pseudo-code should give you a basic idea of how it works (how callback data is cached, etc.). This is not necessarily how it is implemented inside D3DX, but it should help better illustrate the expected behavior of the system.

```
void AdvanceTime( … )
{
    // Callback cache is invalid ?
    if ( bCacheInvalid )
    {
        // THIS IS THE SETUP CASE!
        CachedCallbackData = NULL;        // <-- NULL so handler is not called !!
        CachedCallbackPos  = TrackPos;    // <-- Significant !!
        bCacheInvalid      = false;       // Cache No Longer Invalid

        // Search for next scheduled callback key and cache it
        ProcessCallbacks();

    } // End if must reinitialize cache

    // Natural progression, performed every time ( for every track )
    if ( TrackPos >= CachedCallbackPos ) ProcessCallbacks();

    // All the rest happens here (timer increments and SRT data fetches)
    …
}
```

Here we see what the initial section of the AdvanceTime function might look like. This example only deals with a single track/animation set, but of course in the real D3DX controller there would be a cache for each active track.

The function first tests to see if the cache is invalid. The cache will be invalid if the timer of the track or the global timer has been altered. The cache will also be invalid if this is the first time AdvanceTime has been called with the animation set assigned to the track. If the cache is invalid then the cached callback data pointer is set to NULL and the cached callback event time is simply set to the current position of the track, as discussed earlier. One important thing to notice in the setup phase is that the cached position is set to the value of the track timer *before* the timer has been incremented. That is, the search for callback keys will be performed from the previous track position (not including the passed elapsed time). In this example, the ProcessCallbacks function is then called to search for the next scheduled event on that track and cache it (along with the event context data).

Beyond the setup phase we see a line of code that will be called with every call to AdvanceTime. It compares the current track position to the currently cached callback position. If the track position is greater, then it calls ProcessCallbacks to execute the callback function and fetch the next event in the schedule. Note that ProcessCallbacks is actually a multi-purpose function. When called from the setup phase, it simply searches for the next scheduled event. When called from the compare line that happens every time, it is called when the event has been reached and its callback needs to be handled. The ProcessCallbacks function is shown next:

```
void ProcessCallbacks()
{
   ULONG Flags = (CachedCallbackData == NULL) ? 0 :
                                 D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION;

   // Call the callback handler if anything is available
   if ( CachedCallbackData )
        CallbackHandler->HandleCallback(TrackIndex,CachedCallbackData);

   // Get the next callback (notice how we always pass in the last
   // callback pos, not track time)
   TrackSet->GetCallback( CachedCallbackPos, Flags,
                          &CachedCallbackPos, &CachedCallbackData);
}
```

First let us examine what this function will do if called from the setup phase of the AdvanceTime function. The first thing it will do is set the flags value to 0 because the cached callback data was set to NULL when the cache was invalid. The next line is skipped because the cached callback data will be NULL. The final line calls the animation set's GetCallback function. As the first parameter it passes the currently cached time. Because this was called from the setup phase inside AdvanceTime, the cached callback time is actually the current track position. We also pass in 0 as the flags parameter because, as discussed earlier, when the cache is invalid, we do not want to exclude the initial time from the key search. If there is a key scheduled for the current track position, we want to retrieve and cache it. Finally, notice that the CachedCallbackPos and CachedCallbackData are passed into the handler function. On function return they will contain the time of the next scheduled event on the track and any context data associated with it. At the end of the setup phase, we have a new cached position which is

no longer the current track timer (most of the time) and it will be the next event that needs to be executed.

This function is also called when the current track position reaches the currently cached key time, and it behaves a little differently in this scenario. When called during the natural progression of the AdvanceTime function, it means a cached callback key time has been reached and needs to be handled. We can see that in this case, cached callback data will exist and therefore the flags will be set to exclude the initial position in the next search for a future key. Because there is callback data, we know that we must have called this function previously and a cached event needs to be executed. So in the next line we execute the callback handler function (more on this in a moment). The final line is called to retrieve a new next event time for caching purposes. However, you should notice that when this function is called this time, the cached time is not the track position. The input time to GetCallback is the time of the event that was just handled. This way, even if the track position is many seconds in front, we do not skip any events between the last cached event and the current time.

> **Note:** The pseudo-code we are examining has been greatly simplified. For example, in ProcessCallbacks we are assuming that if there is no callback data defined then no handler needs to be called. But in fact, it is quite possible to register a callback key and assign it no context (i.e., set its callback data to NULL). There is no requirement that you always include context data, and indeed you might not need to send data into your own callback functions. So in truth, our pseudo-code design is slightly incorrect, but hopefully the general concept of key caching is clear.

We have seen how to register keys with an animation set and how the animation controller uses the animation set's GetCallback function to schedule upcoming events on a track. We have also looked at how each callback key can have user-defined data which can be passed to the callback function when the event is triggered. We even have some concept of how the AdvanceTime method executes the callback function. What we have yet to discuss is how we set this application defined callback function. This will be the purpose of our next section.

10.10.3 Setting the Callback Function

When we discussed the AdvanceTime function earlier in the chapter, we deliberately neglected to explain the second (optional) parameter. This parameter will now be examined as it is the means by which we inform the AdvanceTime function about the callback function that should be executed when a callback event is triggered. Recall that the AdvanceTime function definition is:

```
HRESULT AdvanceTime
(
    DOUBLE TimeDelta,
    LPD3DXANIMATIONCALLBACKHANDLER pCallbackHandler
);
```

As you can see, if a second parameter is passed into this function, it should be a pointer to a class of type ID3DXAnimationCallbackHandler. If this parameter is set to NULL then the function will not attempt to process any callback code.

If we take a look at the d3dx9anim.h header file, we will see that ID3DXAnimationCallbackHandler is an interface with a single method:

// Excerpt from d3dx9anim.h

```
#define INTERFACE ID3DXAnimationCallbackHandler

DECLARE_INTERFACE(ID3DXAnimationCallbackHandler)
{
      STDMETHOD(HandleCallback)(THIS_ UINT Track, LPVOID pCallbackData) PURE;
};
```

This interface is similar to the ID3DXAllocateHierarchy interface (Chapter Nine) in that it does not have an associated COM object. It is just an abstract base class that is exposed so that applications can derive their own classes from it. Your application will never be able to instantiate an instance of ID3DXAnimationCallbackHandler, so it must derive a class from it that implements the HandleCallback function. Just as the D3DXLoadMeshHierarchyFromX function is passed a pointer to an ID3DXAllocateHierarchy derived class (which it uses to callback into the application during hierarchy loading), an instance of an ID3DXAnimationCallbackHandler derived class can be passed into the AdvanceTime call. Its only method (HandleCallback) will be called by the AdvanceTime function when a callback event is triggered. As you can see, the function accepts the index of the track on which the callback event was triggered and a void pointer for the callback key context data.

Because you are responsible for implementing the HandleCallback function in your own derived classes, you can program your code to do anything you require. For example, let us continue building on our previous example where we registered three callback keys to play sound effects. We will begin by deriving our own class from ID3DXAnimateCallbackHandler and implement the HandleCallback method. In this example, we will call our derived class CSoundPlayer:

```
class CSoundPlayer: public ID3DXAnimationCallbackHandler
{
    public:
    STDMETHOD(HandleCallback) (THIS_ UINT Track, LPVOID pCallbackData);
};
```

Now we will implement the HandleCallback function so that the name of the .wav file passed into the callback function (as pCallbackData) can be passed into a function that plays a sound.

```
HRESULT CSoundPlayer::HandleCallback( UINT Track, LPVOID pCallbackData )
{
    // cast the past string name back into a TCHAR pointer
    TCHAR * WavName = (TCHAR*) pCallbackData;

    // Play the sound
    PlaySound( WavName , NULL , SND_FILENAME);
}
```

Here we see that the name of the .wav file (pCallbackData) is cast back to a TCHAR pointer. It is passed into the Win32 PlaySound function to load and play the sound. (To be sure, this is not the optimal way to play sounds in your games; this is meant only to serve as a simple callback example).

If you have multiple callback events registered on different tracks, there is a very good chance that not all of these callback keys will simply want to play back sound effects. In this case, your callback function would need to account for a wider range of functionality. One approach might be for your context data to point to an application defined structure instead just of a sound filename. The first parameter of this structure might be a flags member that you define to identify the event type to be processed (PlaySound, StopSound, FadeOut, etc.). Your callback function would simply cast the context data pointer back to the correct structure pointer and examine the flags to determine its course of action.

Finally, with your callback handler class written, you simply have to instantiate an instance of it before calling the AdvanceTime method:

```
void UpdateAnimation( double ElaspedTime )
{
   CSoundPlayer Callback;
   pController->AdvanceTime( ElapsedTime , &Callback );
}
```

10.10.4 Getting and Setting Callback Keys

The ID3DXKeyframedAnimationSet interface exposes four methods relating to the callback system which we have not discussed. Please note that these functions are implemented for the application's benefit only; they are not part of the base ID3DXAnimationSet interface and they will never be called by the animation controller. Thus, the controller will never expect them to be implemented in any custom animation set classes that you develop.

The first method allows the application to query the number of callback keys that are registered with the animation set:

```
UINT GetNumCallbackKeys(VOID);
```

Next, while there exists no way to add additional callback keys to the animation set once it has been created (recall that this memory is allocated at animation set creation time), a function is provided that allows the application to alter the value of a callback key if we know its index. The index is simply the number of the key in the internal callback key array. In our earlier example, we created an animation set with three callback keys; their indices will be 0, 1, and 2 respectively.

In order to set the value of a pre-existing key, we pass the index of the key we wish to change along with the new key information into the SetCallbackKey function.

```
HRESULT SetCallbackKey
(
    UINT Key,
    LPD3DXKEY_CALLBACK pCallbackKeys
);
```

We examined the D3DXKEY_CALLBACK structure earlier in the chapter, and saw that it contains a timestamp and a void pointer that is used to pass application data to the callback handler.

Just as our application can set the value of a callback key, we also have the ability to query key details. To do so, we can use the GetCallbackKeys method and pass in a valid key index and the address of a D3DXKEY_CALLBACK structure. If the index is a valid key index, the key information will be copied into the passed structure:

```
HRESULT GetCallbackKey
(
    UINT Key,
    LPD3DXKEY_CALLBACK pCallbackKeys
);
```

Finally, the application can also request a copy of all callback keys registered with the animation set. The GetCallbackKeys function must be passed a pre-allocated array of D3DXKEY_CALLBACK structures large enough to hold all key information in the animation set. Therefore, before calling this function, you will usually call GetNumCallbackKeys first so that you know how much memory is needed for this array.

```
HRESULT GetCallbackKeys
(
    LPD3DXKEY_CALLBACK pCallbackKeys
);
```

The following code shows how one might use this method to extract all callback keys from an animation set. It is assumed that the animation controller and the animation set are already valid and that the animation set contains a number of callback keys.

```
// Get number of keys in anim set
NumKeys = pAnimSet->GetNumCallbackKeys();

// Allocate array of the correct size
D3DXKEY_CALLBACK *pKeys = new D3DXKEY_CALLBACK[NumKeys];

// Fetch a copy of the keys into our array
PAnimSet->GetCallbackKeys( pKeys );

// Do something with keys here
…
…

// When finished release key array
delete [] pKeys;
```

Conclusion

We have now thoroughly covered all of the fundamentals of mesh hierarchies and animation using DirectX. While the animation controller documentation is a little sparse in the SDK, it should be clear that D3DX provides us with access to a very powerful and relatively easy to use set of animation interfaces.

Before moving on to skeletal animation and skinning in the next chapter, we recommend that you work through the source code to the lab projects accompanying this chapter. The next chapter will make use of use many of the techniques we have examined here, so it would be wise to read through the workbook and make sure that you are comfortable with these topics.

Lab Project 10.1 is an application that demonstrates DirectX animation at its simplest -- loading an animated X file and playing it back. The demo will teach us how to work with and render the frame hierarchy returned to us by the D3DXLoadMeshHierarchyFromX function, as well as how to use the animation controller to play the animation back. We will take this opportunity to build out our CActor class code from the previous chapter by adding the methods exposed by the D3DX animation system. Moving forward, our CActor class will become a very useful class -- it will wrap the loading and rendering of frame hierarchies and expose functions for loading and playing back animations that affect those hierarchies.

In Lab Project 10.2 we will delve even deeper into the D3DX animation interfaces and build ourselves a very handy tool. Given an X file with a single animation set, our new tool will allow us to break that animation set into multiple named animation sets and then resave the data back out to a new X file.

Again, it is very important that you understand everything we have covered so far in the course. You certainly need to be very comfortable with hierarchies and how they are animated before continuing on to the next lesson concerning skeletal animation. On the bright side, you should be pleased to know that most of the hard work is behind us now. Skeletal animation and skinned meshes are really not going to be much more complicated than the material we learned about here. What we will discover in the next chapter is that the frames of the hierarchy will simply be perceived as the bones of a skeleton with often a single mesh (although there can be multiple meshes) draped over the entire hierarchy. So, once you have tackled the lab projects in this chapter, you will be ready to move straight on to the next lesson.

Workbook Ten: Animation

Lab Project 10.1: Adding Animation to CActor

In Lab Project 9.1 we implemented a CActor class that encapsulated the loading and rendering of hierarchical multi-mesh representations. In this lab project we are going to continue to evolve CActor by adding animation support for its internal frame hierarchy. As each actor represents a single hierarchy loaded from an X file, the actor will become the perfect base for modeling game characters as well as other animated entities. Each character is contained in its own frame hierarchy inside the actor and will have its own animation controller to play back its animations via the CActor interface. We must also remember that a single CActor object could also be used to animate an entire scene if all the meshes and animations in that scene were loaded from a single hierarchical X file.

In the Chapter 10 textbook, we learned that D3DX exposes an easy-to-use comprehensive animation system. We will now take the information we learned in that lesson and wrap it up inside of our CActor class. Lab Project 10.1 will use the new extended CActor object to load an animated scene from an X file and play it back in real-time. Figure 10.1 shows the code from lab project 10.1 in action.

Figure 10.1

The X file we will load contains a small futuristic scene with many moving meshes, including a space ship that takes off from a launch pad and flies off into the distance. There are also many other moving meshes such as revolving radar dishes and hanger doors that open and close. In this first simple example, a single X file will be loaded by our actor. The file will contain the entire scene and all animation data in a single animation set. Therefore, a single actor will contain all objects in the scene that you see in Figure 10.1.

Because a single animation set will contain the animations for all objects in our scene, playing back this animation could not be easier. The CActor::LoadActorFromX function will call the D3DXLoadMeshHierarchyFromX function to load the X file. This function will automatically create the animation controller and its animation set before returning it to CActor for storage. (The animation controller will be a new member variable that we will add to CActor.) The controller will then be used by the CActor member functions to play the animated scene in response to application requests.

Because the D3DXLoadMeshHierarchyFromX function will (by default) automatically assign the last animation set defined in the X file to the first track on the animation mixer, when the D3DXLoadMeshHierarchyFromX function returns the newly created controller back to the actor, we have absolutely no work to do. The only animation set defined in the X file in this example will have been assigned to the first mixer track on the controller and is ready to be played immediately. All our CActor has do at this point, is call the controller's AdvanceTime method with every requested update to generate the new frame matrices for the hierarchy. For this reason, Lab Project 10.1 is an excellent initial example of using the animation system. It clearly demonstrates the ease with which cut scenes or scripted sequences can be played out within a game engine. Notice that as the animation sequence plays out, the player still has freedom of movement within the scene. This allows the player to feel part of a dynamic scene instead of simply watching a pre-rendered sequence.

Although Lab Project 10.1 will only use a very small subset of the D3DX animation system's overall features, we will still implement all the animation functionality exposed by D3DX in CActor in this project. This will prevent us from having to continually revisit CActor later in the course when we wish to use animation features that have not previously been coded. This will mean of course that we will have to add many animation methods to our CActor interface so that an application can access all of the features offered by D3DX. For example, our CActor must provide the ability for an application to assign animation sets to tracks on the mixer, configure the properties of those mixer tracks, and allow the application to reset the global timer of the animation controller. The CActor interface must also expose methods that allow the application to register events with the event sequencer on the global timeline as well as expose the ability for scheduling of callback keys for a given animation set. While this sounds complicated, most functions will be simple wrappers around their ID3DXAnimationController counterparts. Once we have this task out of the way, we will have a CActor class that is reusable in all future applications that we develop. This will be a powerful tool indeed and will allow us to add animated objects to our scenes with ease in the future. Furthermore, we will need this animation system in place in the following chapter when we move on to the subject of character skinning and skeletal animation (something else that will be managed by our CActor class).

Expanding the Limits of the Controller

Another feature we would like our actor to expose to the application is the ability to easily upgrade the maximum limits of the controller. In the textbook we learned that the maximum limits of the animation controller must be specified at controller creation time. This is certainly a restraint we could put up with if we were always manually building our animation controllers, but when using the D3DXLoadMeshHierarchyFromX function, the controller is created for us and we are afforded no control over the maximum values. For example, the controller created by

D3DXLoadMeshHierarchyFromX will always have two mixer tracks. This may not be enough to satisfy the blending desires of our applications. Additionally, the maximum number of animation sets that can be simultaneously registered with the controller will always be equal to the number of animation sets defined in the X file. Once again, this is often acceptable, but sometimes our application may wish to programmatically build additional animation sets once the X file has been loaded, and then register these new sets with the controller. The controller returned from D3DXLoadMeshHierarchyFromX would have no room for these animation sets to be registered, so we will have to account for this limitation.

We will address this problem by exposing a method called CActor::SetActorLimits. With this function, we can specify the exact number of animations, animation sets, and mixer tracks that we would like our actor's animation controller to support. This function is very flexible because you can call it before or after the X file has been loaded. If you call it before you have call CActor::LoadActorFromX, then the maximum limits you pass in will be stored in the actor's member variables. When the call to CActor::LoadActorFromX is finally made and the animation controller is created, the function will automatically clone the D3DX-generated animation controller into one which matches the desired maximum limits. When CActor::LoadActorFromX returns control back to the application, the actor will contain its frame hierarchy with all animation data intact and an animation controller with the desired maximum limits. We can also call the CActor::SetActorLimits method even after the X file has been loaded and it will still work fine. If the X file is already loaded, the SetActorLimits method will simply clone the current animation controller being used into one with the desired maximums specified by the application.

> **Note:** When we clone the animation controller, we can specify different creation parameters from those used in the original (e.g., a higher number of mixer tracks). All the animation data stored in the original controller will be copied into the cloned controller, ensuring data integrity. Our CActor::LoadActorFromX and CActor::SetActorLimits methods use cloning to allow the application to override the default limits of the animation controller created by the D3DX loading function. This also allows our application to expand the limits of the actor's controller at any time, even when the application is running.

The following code shows how an application might use the CActor class to load an animated X file and then set the limits of the controller:

```
// Create Actor
CActor *pActor=new CActor;

// Register scene call back function to manage mesh resources
pActor->RegisterCallback( CActor::CALLBACK_ATTRIBUTEID, CollectAttributeID,this );

//Load an X file into the actor. All meshes in the managed resource pool
pActor->LoadActorFromX( "MyXFile.x", D3DXMESH_MANAGED, m_pD3DDevice );

// Set the controller so that it can handle AT LEAST 15 animation sets and 10
// blending tracks and capable of registering 150 sequencing events
pActor->SetActorLimits( 15 , 10 , 0 , 150 , 0 );
```

As the previous code demonstrates, changing the limits of the actor, even after its controller has been created, is perfectly safe. We will look at this function in detail later. For now, just know that if we pass in zero for any of the parameters, that property will be inherited from the original controller during the cloning process. For example, the third parameter to this function is where we can specify the number of

animation outputs we would like the actor's controller to handle. This will have already been set in the original controller (by D3DX) such that it can at least animate all the animated frames in the hierarchy. This is usually exactly what we want it to be. Unless we are adding additional frames to the hierarchy programmatically, we will normally not want to alter this value. Therefore, by passing in zero for this property, the SetActorLimits method knows that the new controller it is about to clone should have its maximum animation outputs value remain the same as the controller from which it is about to be cloned. This is useful because the application will not be forced to query the number of animation outputs currently being managed by the actor's controller, simply to pass that same value back into the SetActorLimits method unchanged.

The final parameter to this function is where we can specify the maximum number of callback keys that can be registered with any animation set. We will discuss why this is needed in the following section. It is also worth noting that the application can use the SetActorLimits method whenever it likes. For instance, in the above example we set the maximum number of mixer tracks to 10. However, later in the application we might decide to expand that to 15. We could do so simply by calling the function again with 15 instead of 10 as the second parameter. We could pass zero in for all the other parameters so that the newly cloned controller will inherit all the other properties from the original controller.

The Actor Exposes a One-Way Resize

For reasons of safety and ease of use, we made the design decision that the SetActorLimits method should only ever clone the controller if any of the specified maximum limits exceed those of its current controller. The function will never clone the controller if we specify maximum extents which are less than the current controller's maximums. There are a number of reasons we decided to take this approach.

First, we do not wish to burden the application with having to write code to query the current maximum extents of the controller. It is much more convenient if we can just specify the properties we need (such as the number of mixer tracks) without having to be aware of all the other properties that were setup automatically (e.g., the number of animation outputs). Second, we could accidentally break the controller's ability to animate the data loaded from the X file if we downsized its maximum extents such that it could no longer properly use the information stored in the original X file. For example, if the original controller created by D3DX needed to animate 50 frames in the hierarchy, it would have been created with the ability to manage 50 animation outputs. If we then called the SetActorLimits method and incorrectly set this parameter to some value less than 50, information would be lost in the clone operation and some or all of the frames in the hierarchy that should be animated would no longer be. This would essentially break the entire animated representation created by the artist. In order to avoid this, the application would first need to query the current number of animation outputs being used by the controller and would need to pass no less than this value into the SetActorLimits method. With our 'expand-only' philosophy, this entire burden is removed. We do not even have to know about the properties of the mixer that we have no interest in. We can simply specify that we require no less than a certain value for a given extent.

In the expand-only system, the SetActorLimits method will never clone an animation controller that is incapable of animating the data in the original controller, regardless of the values passed in by the application. This allows the application to call this method with any value without breaking the actor's ability to animate the original data loaded from the file. While this does mean that there may be times when the maximum limits of the controller may be superfluous to your applications requirements, this is often very rare, since the original animation controller will be created with the bare minimum capabilities needed to animate the original X file data. Therefore, you will usually only want to extend these capabilities and not reduce them. One exception to the rule might be if your application only wishes to use a single mixer track. We know from the accompanying textbook that the animation controller created by D3DXLoadMeshHierarchyFromX will always have at least two mixer tracks. In this case, the animation controller would have a mixer track that you have no intention of using so you could downsize. However, the memory taken up by a single wasted mixer track is certainly acceptable given the ease and safety at which the actor can now be used.

A Flexible Callback Registration Mechanism

One of the more frustrating limitations of using the controller's callback system is that the memory for the callback keys must be allocated at animation set creation time. You will recall from the textbook that when creating an animation set manually (using the D3DXCreateKeyframedAnimationSet function), we must specify how many callback keys we would like to register with the animation set and pass in a pointer to an array of callback keys. These callback keys are then allocated and stored inside the animation set. While this is a limitation we can live with when creating animation sets manually, we know only too well that when using the D3DXLoadMeshHierarchyFromX function, the animation controller and all its animation sets are created for us automatically. As it is very unlikely that you would have callback keys stored in your X file, we can assume that under most circumstances the application will want to register callback keys with the controller *after* the X file has been loaded. At this point however, the animation controller will have been created along with its animation sets (with no callback key data assigned to them). We cannot simply add callback keys to an animation set at this time, because the memory for our callback keys will not have been allocated when the animation set was created by D3DX. D3DX would have had no way of knowing that we intended to register anything at all. Suffice to say, cloning will come to the rescue here as well.

The system we designed to handle the post-load registration of callback data does not just involve data cloning. It also incorporates an application-defined callback function which will be registered and called by the actor to allow the application to fill an actor owned temporary callback key array. This callback function will be called by the actor in the CActor::LoadActorFromX function just after the X file has been loaded. The actor will pass the application defined callback function an array of callback keys which the function can fill with callback data. When the callback function returns, the actor will have an array of callback keys for a given animation set. It will now need to replace the respective animation set that was loaded by D3DX with a clone which has room for the callback data. We will not be cloning the animation controller but will instead manually create copies of the animation sets ourselves. That is to say, for each animation set currently registered with our controller, we will create a new animation set with the correct number of callback keys reserved. When we create these animation set copies, we will pass in the callback data that the application defined callback function stored in our temporary callback

key array. Once we have created copies of all the animation sets with the callback data registered, we will un-register and free the original animation sets and register the new ones (with the callback data) in their place.

As callback keys are defined on a per-animation set basis, the application defined callback function will be called once for each animation set loaded from the X file. This will allow the application to register callback data with any of the loaded animation sets. All this will happen inside the CActor::LoadActorFromX function, abstracting away the application from the entire process. All the application has to do is implement the callback function and register it with the actor prior to calling CActor::LoadActorFromX.

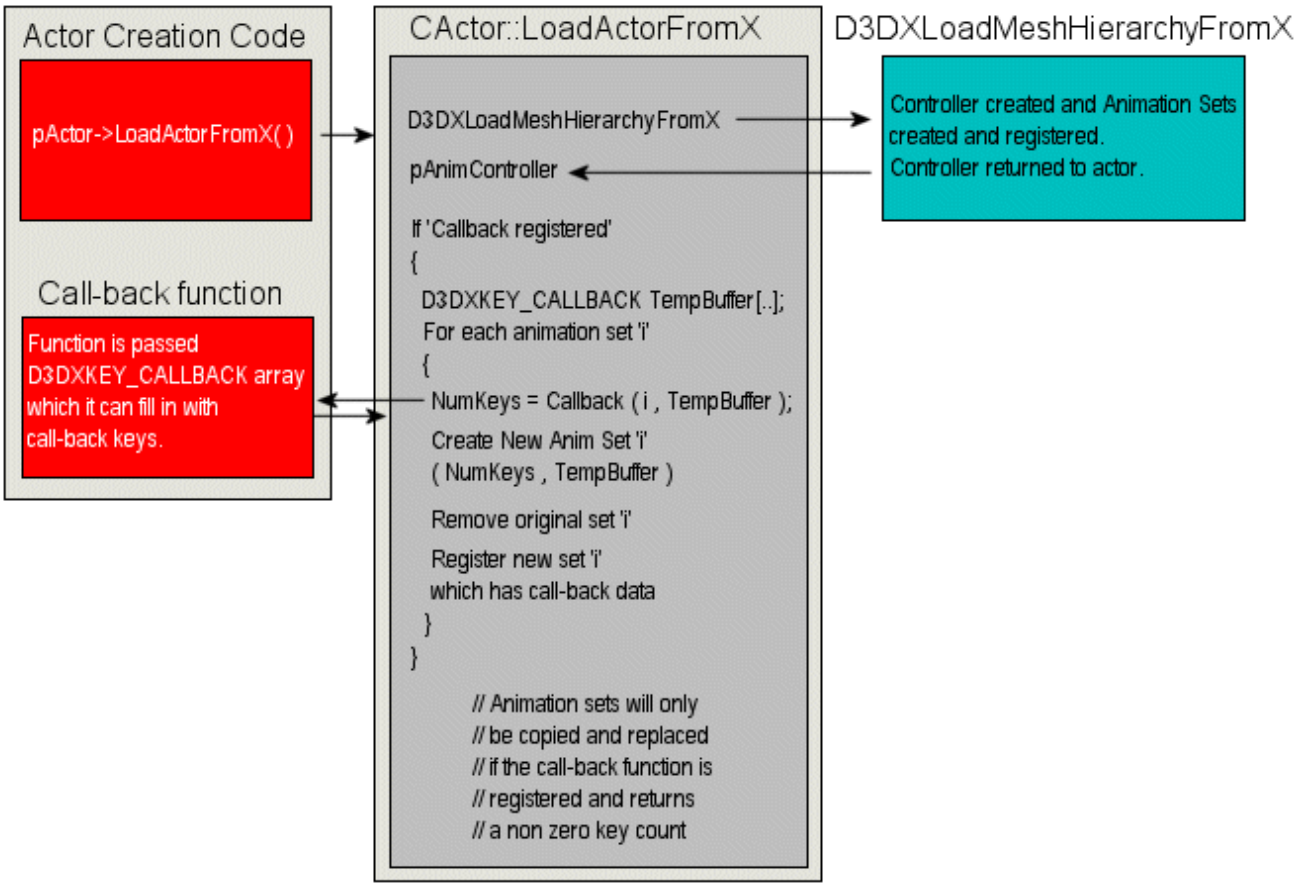


Figure 10.2

Figure 10.2 demonstrates the type of interaction that we will establish between the actor and the application. The application calls CActor::LoadActorFromX, which in turn calls D3DXLoadMeshHierarchyFromX to load the X file. If the X file contains animation data, the actor will be returned a pointer to an animation controller. At this point, if the application has registered a callback key callback function with the actor, it will allocate a temporary array of callback keys. It will then loop through each animation set registered with the controller and call the application defined callback function. This function will be passed the temporary callback key array which it can use to fill with callback keys for that animation set. The function should return an integer describing the number of callback keys it placed in this array. If this is non-zero, the actor will create a new animation set, passing

in this array of callback keys at creation time. The new animation set will not yet be attached to the controller at this point but will contain all the callback keys the application callback function specified for it. The next task will be to copy over all the animation data from the original animation set into the new animation set so that all animation data is maintained. At this point the new animation set will be a perfect clone of the original, with the exception that it now has the callback keys contained within. Finally, we un-register the old animation set from the controller and replace it with the new one. This will be done for each animation set registered with the controller, giving the application the chance to replace any animation sets that need callback key data.

The application defined callback function is also passed a pointer to the original animation set interface. This allows the callback function to check the name of the animation set for which it is being called so that it can determine if it wishes to register any callback keys for this set.

In our implementation, we decided to let the actor allocate the temporary callback key array that is passed into the callback function. One might wonder why we do not just let the callback function return a pointer to an application allocated array. In fact, this would present some problems. First, if the application allocated the array and returned a callback key array pointer to the actor, the programmer might be tempted to generate this array on the stack in the callback function. When the callback function returns the pointer to the actor, the stack information will be destroyed and the actor will have an invalid pointer. Second, even if the callback array is dynamically allocated on the heap in the callback function and returned to the actor, the application will need to implement a cleanup system where this memory is freed when the actor is deleted. This would need to be a little bit like the DestroyMeshContainer callback method that we used during frame hierarchy destruction. We certainly cannot allow the actor to free this application allocated memory in its destructor since it did not allocate this memory itself. It would not know for sure how the memory was allocated and therefore how it should be freed (e.g., we would not want our actor to use ‘delete’ instead of ‘free’ if the application had allocated the key array using malloc).

This is all a bit of a pain for the sake of a simple temporary buffer that is only used to communicate callback keys from the application to the LoadActorFromX function. After all, this temporary buffer is no longer needed once the new animation set has been created because the animation set creation function makes an internal copy of the callback data passed in. Therefore, what we decided to do is simply let the *actor* allocate a temporary buffer which is reused with each call to the callback function. The callback function can simply fill this buffer, which is then automatically accessible to the actor on function return. Furthermore, because the actor allocated this memory, it can release it with confidence after all the callback keys have been registered for each animation set.

Registering a callback key callback function with our actor will be done in exactly the same way we register texture and attribute callback functions. You will recall from the previous lab project that the actor maintains a callback function array (see CActor.h):

```
CALLBACK_FUNC               m_CallBack[CALLBACK_COUNT];
```

Each element of this array defines a single callback function contained in a CALLBACK_FUNC structure (see main.h):

```
typedef struct _CALLBACK_FUNC   // Stores details for a callback
{
    LPVOID  pFunction;          // Function Pointer
    LPVOID  pContext;           // Context to pass to the function

} CALLBACK_FUNC;
```

Each member of this structure contains two void pointers. The first member is where a pointer to the callback function should be stored. The second member is where we can store a pointer to arbitrary data that we would like passed to the callback function. In our projects we always use this to store a pointer to the instance of the CScene class in which the callback function is a static method. This is so the callback function can access non-static members.

You will recall that we store information in this array using the CActor::RegisterCallback function. We used this same function to store texture and attribute callback functions in the actor's callback function array in previous lab projects (see CActor.h):

```
bool RegisterCallback  ( CALLBACK_TYPE Type, LPVOID pFunction, LPVOID pContext );
```

This function (which we covered in a previous lesson) takes the passed function pointer and context pointer and stores them in a CALLBACK_FUNC structure in the actor's callback array. You will hopefully recall that the first parameter of this function should be a member of the CALLBACK_TYPE enumerated type which is defined within the CActor namespace:

```
enum CALLBACK_TYPE
{
    CALLBACK_TEXTURE = 0,
    CALLBACK_EFFECT = 1,
    CALLBACK_ATTRIBUTEID = 2,
    CALLBACK_COUNT = 3
};
```

This enumerated type describes the index in the actor's callback array where a particular callback function should be stored. For example, if we pass CALLBACK_TEXTURE into the CActor::RegisterCallback method, this pointer and context will be stored in the first element of the array. Using this enumerated type, the actor can quickly test the index of its callback array to see if a particular callback function has been registered by the application.

Previously, we only needed callback functions to parse textures and materials during X file loading, but now we need to add a new member to this enumerated type. We must also increase the CALLBACK_COUNT member so that we make room at the end of the actor's callback function array for a new callback function type. This callback function will be called by the actor (if it has been registered) to collect callback keys for animation sets during the loading process.

The updated CALLBACK_TYPE enumeration is now defined in CActor.h as:

```
enum CALLBACK_TYPE
{
 CALLBACK_TEXTURE = 0,
 CALLBACK_EFFECT = 1,
 CALLBACK_ATTRIBUTEID = 2,
 CALLBACK_CALLBACKKEYS = 3,
 CALLBACK_COUNT = 4
};
```

If element [3] in the actor's callback function array is non-NULL, it will contain the pointer to a callback keys callback function. Notice that because we have now set CALLBACK_COUNT to 4, the callback function array will be statically allocated in the actor's constructor to contain four elements instead of three.

So, registering our callback keys callback function will be no different from registering any of the attribute callback functions. In fact, the following example shows how the CScene class might allocate an actor, register an attribute callback and a callback keys callback function, before loading the animated X file into the actor. As you will see, it is simply an additional function call. Remember that the 'this' pointer passed into the CActor::RegisterCallback method (as the context parameter) allows the instance of the scene class access to the non-static member variables within the static function.

```
//Allocate a new actor
CActor * pNewActor = new CActor;

// Register an AttributeID call back ( non-managed mode actor )
pNewActor->RegisterCallback(CActor::CALLBACK_ATTRIBUTEID,CollectAttributeID, this );

// Register callback function to add callback keys to the actor
pNewActor->RegisterCallback(CActor::CALLBACK_CALLBACKKEYS, CollectCallbacks, this );

// Load the X file into the actor
pNewActor->LoadActorFromX( "MyXFile.x", D3DXMESH_MANAGED, m_pD3DDevice );
```

In order for the actor to be able to call the callback function, it must be aware of the function's parameter list and return type. Therefore, when we implement callback functions, we must make sure that we follow the strict rules laid down by the actor. When the CActor calls this new callback function, it will use a pointer of type COLLECTCALLBACKS (defined in CActor.h):

```
typedef ULONG (*COLLECTCALLBACKS ) ( LPVOID pContext,
                                     LPCTSTR strActorFile,
                                     LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                     D3DXKEY_CALLBACK pKeys[]
                                   );
```

This is a pointer to a function that returns an unsigned long (the number of callback keys placed into the passed array by the callback function). The function should also accept four parameters: a void pointer to context data (the instance of the CScene class for example), a string (the name of X file will be passed by the actor here), an ID3DXKeyframedAnimatonSet interface pointer (the animation set currently being processed by the actor which we may wish to register callback keys for), and an empty array of callback keys that the function will place its callback key data into if it wishes to register any for the passed set. We must make sure that our collect keys callback function is defined exactly in this way.

Below, we see how the CollectCallbacks method is defined in CScene.h. This is the callback function which will be registered with the actor for the processing of callback key data. Note how it matches the exact function signature described above. The details of the method itself will be discussed in a moment.

```
static ULONG CollectCallbacks ( LPVOID pContext, LPCTSTR strActorFile,
                                LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                D3DXKEY_CALLBACK pKeys[] );
```

Our callback functions must always be static if they are to be methods of classes. They must exist even if no instance of the class has been allocated. The function implementation would then be started like this:

```
ULONG CScene::CollectCallbacks( LPVOID pContext, LPCTSTR strActorFile,
                                LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                D3DXKEY_CALLBACK pKeys[] )
{
   // fill in array here with key data
}
```

Before we start to discuss how to fill in the callback key array, one thing may be starting to trigger alarm bells. How does our callback function know how big the empty D3DXKEY_CALLBACK array (passed in by the actor) is going to be so that it does not try and store more keys than the array has elements? Furthermore, this temporary array has been allocated by the actor before being passed to us. So how does the actor have any idea how much memory to allocate for this array? It would seem to have no means for knowing how many callback keys we might want to register for a given animation set.
But this is not the case. When we discussed the SetActorLimits method earlier, we deferred discussion of its final parameter, simply saying that this was where we specified the maximum number of callback keys that can be registered with a single animation set. This setting is unlike the other properties that we set via SetActorLimits in that it does not alter any properties of the controller -- it simply informs the actor how big to allocate this temporary callback array that will be passed into the callback function during loading.

For example, if we specify (via SetActorLimits) a value of 20 for the maximum callback key count, then a temporary array of 20 D3DXKEY_CALLBACK structures will be allocated in the LoadActorFromX method. This temporary array will be passed into the callback function each time it is called (for each animation set loaded). It stands to reason that we would never want to try and store more than twenty keys in the passed array in this example for a given animation set. As this same temporary buffer is used for key collection for all animation sets, it also stands to reason that the callback function must return the number of key's it placed into the array. If we decide for example, that we would like to register 10 keys for animation set 1 but only 5 with animation set 2, we would set the actor limits so that it had a maximum of 10 callback keys per animation set. When the actor was loaded, the callback function would be called twice. Both times an array of 10 elements would be passed that can be filled by the callback function for each set being processed. The second time the function was called (for animation set 2) the callback function would only use 5 out of the 10 possible array elements. Therefore, the return value of the function correctly informs the actor how many keys out of the possible 10 (in this example) were actually used for a given animation set. This value is then used to reserve space for callback keys

in the newly cloned animation set and the callback data copied. The temporary buffer is no longer needed and can be freed once all animation sets have been cloned.

Of course, the important point is how we might implement such a callback function. While we know that the D3DXKEY_CALLBACK structure is a simple structure that contains only two members (a timestamp and a void pointer to context data), it is exactly in the setting of this context data that certain problems arise which must be overcome. This leads on to our next section.

Safe Callback Data

One of the problems with registering context data with a D3DXKEY_CALLBACK structure occurs when the animation controller is released. Let us say for example, that we implement a callback key callback function that simply sets one callback key for an animation set called “Tank” inside an X file called “MyXFile.x”. We will assume that we wish to assign an arbitrary structure (which we shall call MyStruct) as the context data to this callback key. The members of this imaginary structure are not important -- it might contain sound effect data or a series of properties that need altering on the mixer track when this event occurs -- for this example. A naïve first approach might be to implement the callback registration function like so:

```
ULONG CScene::CollectCallbacks( LPVOID pContext, LPCTSTR strActorFile,
                                LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                D3DXKEY_CALLBACK pKeys[] )
{

    // Is it the right actor
    if ( _tcsicmp( strActorFile, _T("MyXFile.x") ) != 0 ) return 0;

    // Make sure we only assign it to the correct animation set
    if ( _tcsicmp( pAnimSet->GetName(), _T("Tank") ) != 0 ) return 0;

    // Allocate an arbitrary structure
    MyStruct * pData = new MyStruct;
    MyStruct->A = some data goes here
    MyStruct->B = some data goes here
    MyStruct->C = some data goes here
    MyStruct->D = some data goes here


    // store the data object in the call back structure along with the timestamp
    pKeys[0].Time          = 150.0;
    pKeys[0].pCallbackData = (LPVOID)pData;


    // Notify that we have one key
    return 1;
}
```

While this function makes a glaring error (which we will discuss in moment), it does show how the callback function can be used to add a key to a specific animation set. This function, when called by the actor during the loading process, would be returned a single callback key for the animation set.

This function will be called by each actor for which this function is registered. Therefore the first thing we do is test to see whether this function is being called by the right actor. In this example we are only interested in the actor created from the X file called 'MyXFile.x'. As the function is passed the name of the actor (which is the same as the X file it was created from), we use the _tcsicmp method to compare the name of the actor against the string of the X file we are interested in. If there is no match, then this function has been called by an actor that we are not interested in registering callback data for, so we exit.

Next we test the name of the passed animation set against the string "Tank", as we are only interested in registering a callback key for the Tank animation set. This is why it is very useful that the interface of the animation set for which the function is being called for is also passed in by the actor. It allows us to use the ID3DXKeyFramedAnimationSet interface to query the animation set to determine if it is an animation set we are interested in. Remember, although at this point in the function we know we have the actor we are looking for, this function will be called by that actor for every animation set it contains. As we are only interested in one of these animation sets, we perform the string comparison between the name of the animation set passed in and the name of the animation set we are looking for and return from the function if no match is found.

Finally, we allocate some arbitrary structure on the heap, assign it some arbitrary data and assign its pointer to the first callback key in the callback key array. A pointer to this structure will be passed to the callback handler passed into AdvanceTime, which may contain data such as the name of a sound effect to play. In this example we have set the timestamp of the callback key to 150 ticks. What these ticks mean in seconds is dependant on the ticks per second ratio of the animation set. If you wish to find that out so that you can convert the second value into a tick value, there is no problem there. Because the function is passed the ID3DXKeyframedAnimationSet interface, we can call its GetSourceTicksPerSecond method and use the returned value to map from one to the other.

So, we have a good idea now how our callback key callback function works. Obviously, this function might grow quite large if you wish to assign lots of different callback keys to lots of different animation sets during the loading process. But even in that case, the system is quite simple. So what was wrong with the above example?

The problem is tied into the procedure of destroying the actor and its underlying animation controller. When we destroy the actor, the actor releases its controller. The controller in turn releases each of its animation sets until it has all been unloaded from memory. The problem is that when the animation set is released, its callback key array is destroyed also (just as we would expect). However, in the above example, we allocated a structure on the heap inside the registration function and stored its address in the callback key's context data pointer. If the callback key structure is simply released, the only surviving pointer to that structure will be deleted also. We now have no way of freeing the memory of the context data structure that was allocated in the callback registration function. When the application terminates, we will have memory leaks, especially if lots of callback keys are being used which have had structures allocated for their context data in this way. This is an easy problem to fix, but we wish to use a solution that is elegant.

One approach might be to fetch all callback key structures from each animation set inside the actor's destructor before it releases its animation controller. We know that the ID3DXKeyframedAnimationSet interface exposes a GetCallback method to allow the controller to step through its list of callback keys. We can simply use this same function to get at the context pointer of each callback key in the animation set's callback key array. Before releasing the controller, we would loop through each animation set, and for each set, repeatedly call its GetCallback method until we have fetched all of its callback keys. You will recall from the textbook that each call to this method returns the time of the callback key and its context data pointer. We can use these context pointers to delete the structures that have been allocated and assigned to this pointer in the callback registration function.

But there is a problem with this approach too. When we fetch the callback context pointer (via the GetCallback method) we are returned a void pointer to the context data. We have no idea how the application allocated it or even what the pointer is pointing at. It could be the address of a structure or class allocated on the heap, in which case releasing it is exactly what we want to do. On the other hand, it could be the address of a constant global variable that was statically allocated. We would cause an error if the actor tried to release something like that. The actor simply has no idea what the context pointer of a callback key is pointing to and how the data it points to was allocated. Therefore, there is no way it can safely make any assumptions about how to delete it (or even if it should be deleted).

Now you might be thinking that one way around this would be to avoid forcing the actor to release this memory at all. Instead, when we allocate the structure inside the callback function, we could add a copy of its pointer to some CScene array. Therefore, even when the actor is deleted and all callback key arrays have been deleted, the scene object would still have the array of all the context data pointers which could be deleted in a separate pass when the application is shut down. This is not very nice though since we should not be happy with the application having to implement its own container class simply to handle a process that should be performed by the actor. Furthermore, we may wish to assign many different types of structures (as context data) to different callback keys, and they may not all be able to be derived from the same base class. In short, the application may have to maintain multiple arrays of callback key context data pointers for the various types that may be used. This approach is ungainly and simply will not do.

COM to the Rescue

If we think about the way that COM was designed, one of its aims was to eliminate this sort of problematic behavior. COM objects support lifetime encapsulation (i.e., a COM object is responsible for releasing itself from memory when it has no outstanding interfaces in use). All of our problems can be solved if our actor enforces the strict rule that the context data pointer of a callback key must be a pointer to a COM interface. In fact, it does not even have to be a COM object, just as long as its interface is derived from IUnknown. As we know, IUnknown is the interface from which all COM interfaces are derived and it exposes the three core methods: AddRef, Release, and QueryInterface. If our actor knows that the context data pointer for each callback key is a pointer to an IUnknown derived object, it knows that the object must also expose the Release method. Therefore, our actor need not concern itself with what type of object or structure it is pointing to, it can simply loop through each callback key in each animation set, cast the context data pointer to an IUnknown interface pointer, and

call its Release method. This will decrement the object's internal reference count. When the count reaches zero, the COM object will then unload *itself* from memory.

To be clear, we do not need to write a fully functional COM object for this system to work. Rather, we just have to make sure that the classes we allocate and assign as context data are derived from IUnknown and implement the three methods of the IUnknown interface. This is actually a useful thing for us to do because it will also go a long way towards educating you in the construction of a real COM object. We will create a class that is derived from IUnknown and behaves like a COM object in many respects. However, it will not be registered with the Windows registry like a real installed COM component; it will just be a class which we allocate using the new operator. However, this object will maintain its own reference count and delete itself from memory when this reference count reaches zero.

For this task, we will develop the CActionData class, but you can use any class you like just so long as it is derived from IUnknown and fully implements the three core COM methods.

Source Code Walkthrough – CActionData

The first class we will cover will be the CActionData class. It is a simple class and has no association with the CActor class whatsoever. In fact, this class is defined in CScene.h and will be used by our application as the class used to represent callback key context data. The class declaration is shown below:

```
class CActionData : public IUnknown
{
public:
    // Public Enumerators for This Class
    enum Action { ACTION_NONE = 0, ACTION_PLAYSOUND = 1 };

    // Constructors & Destructors for This Class.
             CActionData( );
    virtual ~CActionData( );

    // Public Functions for This Class
    // IUnknown
    STDMETHOD(QueryInterface)(THIS_ REFIID iid, LPVOID *ppv);
    STDMETHOD_(ULONG, AddRef)(THIS);
    STDMETHOD_(ULONG, Release)(THIS);

    // Public Variables for This Class
    Action  m_Action;       // The action to perform

    union                   // Some common data types for us to use
    {
        LPVOID  m_pData;
        LPTSTR  m_pString;
        ULONG   m_nValue;
        float   m_fValue;
    };
```

```
private:
    //-----------------------------------------------------------------------
    // Private Variables for This Class
    //-----------------------------------------------------------------------
    ULONG   m_nRefCount;    // Reference counter
};
```

There are a few very important points that must be made about this class. Firstly, it is derived from IUnknown, which means it can be treated like a normal COM object by the actor when it wishes to release it from memory. As far as the actor is concerned, it will see every valid callback key context data pointer as being a pointer to an IUnknown COM object and will simply call its Release method. Obviously, for this to work, this class must implement the QueryInterface, AddRef and Release methods set out in the base interface. Notice at the bottom of the declaration there is a private member variable called m_nRefCount. This will be used to represent the reference count of the object which will be incremented and decremented with calls to its AddRef and Release methods respectively. This will be set to 1 when the object is first created. The Release method will have additional work to do if it decrements the reference count of the object to 0. When this is the case, it means no code anywhere wishes to use this object any more, and it will delete itself from memory. We will look at this function in a moment.

So how does this class work? It is rather simple and can be easily extended to represents all sorts of callback key data types. As you can see, the class has defined a union so that at any one time it can store a void pointer, a string, an unsigned long, or a float. That covers virtually all bases when we consider that we could use the void pointer to point to arbitrary structures. As this is a union, all members share the same memory, so only one can be active at any one time. Which one is active depends on the setting of the m_Action variable, which can be set to a member of the Action enumerated type. Currently, this has only two members: ACTION_NONE and ACTION_PLAYSOUND. You can add different modes to this class and the code to properly release the memory for them in its destructor. For example, in Lab Project 10.1 we use the ACTION_PLAYSOUND action when we register callback key context data. We know that in this mode, the CActionData object will expect the name of a wave file to be stored as a string (i.e., the m_pString member of the union will be used). Therefore, in the destructor, if we test the m_Action member variable and find that it is set to ACTION_PLAYSOUND, we know that this represents sound effect context data and we can free the memory of the string. Obviously, if you added your own types, you would know what data it used, how it was allocated, and therefore how it must be de-allocated when the object destroys itself.

Before looking at the code to this object it will be helpful to take a look at the callback registration function implemented in CScene.cpp in Lab Project 10.1. This callback function is called by the actor when the X file is loaded. We use our new CActionData class to register two sound effect callback keys with an animation set called “cutscene_01”. This should give you a good idea about how the CActionData class is used before we discuss the code.

```
ULONG CScene::CollectCallbacks(     LPVOID pContext,
                                    LPCTSTR strActorFile,
                                    LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                    D3DXKEY_CALLBACK pKeys[] )
{
    CActionData * pData1 = NULL, * pData2 = NULL;
```

```
    // Bail if this is not right actor or right animation set
    if ( _tcsicmp( strActorFile, _T("Data\\AnimLandscape.x") ) != 0 ) return 0;
    if ( _tcsicmp( pAnimSet->GetName(), _T("Cutscene_01") ) != 0 ) return 0;

    // Allocate new action data objects for both callback keys
    pData1 = new CActionData();
    if ( !pData1 ) return 0;
    pData2 = new CActionData();
    if ( !pData2 ) { pData1->Release(); return 0; }

    // Setup the details for both callback keys (take off sound, and flying sound)
    pData1->m_Action  = CActionData::ACTION_PLAYSOUND;
    pData1->m_pString = _tcsdup( _T("Data\\TakeOff.wav") );
    pData2->m_Action  = CActionData::ACTION_PLAYSOUND;
    pData2->m_pString = _tcsdup( _T("Data\\Flying.wav") );

    // Store the two data objects at the correct times
    pKeys[0].Time          = 0.0;
    pKeys[0].pCallbackData = (LPVOID)pData1;
    pKeys[1].Time          = 5.0 * pAnimSet->GetSourceTicksPerSecond();
    pKeys[1].pCallbackData = (LPVOID)pData2;

    // Notify that we have two keys
    return 2;
}
```

When called by an actor, this function will return zero if the actor that called it was not loaded from a file called ‘AnimLandscape.x’ or if the animation set passed in is not called “Cutscene_01”.

If all goes well, then it will allocate two new CActionData objects because in this example we want to register two callback keys to play sound effects at some point along the animation set’s timeline. We then set the details of each CActionData object. The reference count of each CActionData object will be 1 at this point (via its constructor). We set the m_Action member of both to ACTION::PLAYSOUND. For the first CActionData object we set its string pointer member variable to point to a string containing the name of the wave file that should be played when this event is triggered (“Data\\TakeOff.wav”). We do the same for the second CActionData object, only this time we set the name of the wave file to “Data\\Flying.wav”. Notice that we are using the _tcsdup function which duplicates the string passed in. This means that the string data must be freed when the CActionData object is deleted in its destructor.

Once we have the context data stored in the CActionData objects, we assign them to the context data pointer for the first two keys in the passed D3DXKEY_CALLBACK array. The first event is triggered 0 seconds into the animation set’s period and the second event is triggered at 5 seconds. Finally, we return the count of two so that the actor knows that it must clone this animation set into a new one which has room for two callback keys.

> **Note**: At 0 seconds animation set time, the controller will call a callback function (more on this in a moment) which it uses to handle callback keys. This callback function will be passed the CActionData object associated with that key. The callback function can then extract the string “Data\\TakeOff.wav” from the passed object which informs it of the sound to play. At 5 seconds, this same callback function will be called again, only this time, when the string is extracted from the CActionData object, it contains “Data\\Flying.wav” causing a different sound to be played. We will discuss how the callback function should

be written later. At the moment, we are just focusing on a secure way to allocate and store the callback key context data inside the callback registration function.

You will see later that contrary to how we usually do things, when the CActionData object pointers are returned to the actor, the actor does not increment their reference count. This is because the callback function should really be seen as an extension of the actor itself, as it is called only by the actor during the loading process. Therefore, we can think of this function as allocating the object on behalf of the actor before storing its pointer in an actor-owned data structure (the callback key array). This is because we wish the reference count of each CActionData object to be exactly 1, so that when the actor releases the controller, we can fetch each callback key of each animation set and then cast its context data pointer to IUnknown. We can then call IUnknown::Release which will cause the reference count of the CActionData object to drop to 0 resulting in its unloading itself from memory. If the actor also incremented the reference count when the function returned, the CActionData objects would not destroy themselves on actor destruction because their reference counts would fall from 2 to 1 instead of from 1 to 0.

CActionData::CActionData()

The constructor sets the data area to NULL and the action of the object to ACTION_NONE. This is the default action of a newly created object which basically states that the object has no valid data which needs to be released on destruction. We saw in the above code that the callback function assigned the action to something meaningful (ACTION_PLAYSOUND) as soon as the object was allocated.

It is very important to note that we set the internal reference count of the object to 1. We wish this class to behave like a real COM object so we know that if the constructor has been called, then an instance of this class has been allocated and there will a single pointer to the object currently in use by the application.

```
CActionData::CActionData( )
{
    // Setup default values
    m_Action = ACTION_NONE;
    m_pData  = NULL;

    // ***********************************
    // *** VERY IMPORTANT
    // ***********************************
    // Set our initial reference count to 1.
    m_nRefCount = 1;

}
```

CActionData::AddRef

Just like all COM objects, our object should implement an AddRef method which increments the reference count of the object. An application should call this method whenever it makes a copy of a pointer to an object of this type. We never make any additional copies of our CActionData object in our

lab project since the only pointer to the object is stored in the D3DXKEY_CALLBACK structure. However, if your application does intend to store a copy of a pointer to this object somewhere, it should call AddRef after it has done so. This is how the object knows how many interfaces are in use by external code. We certainly would not want the object to unload itself from memory while an external code module still thought it could use its pointer to this object.

```
ULONG CActionData::AddRef( )
{
    return m_nRefCount++;
}
```

CActionData::Release

For any external code or objects that are using our object, the Release method is its way of letting us know that it is no longer interested in using it. We are used to calling the Release method of COM interfaces by now so this concept is nothing new. What is new however is the implementation of a Release method which must decrement the internal reference count and instruct the object to delete itself if the reference count reaches zero. It should now be clear why using an interface to a COM object that has been released will cause your code to crash. After the reference count reaches zero, the object and all its data simply do not exist any more.

```
ULONG CActionData::Release( )
{
    // Decrement ref count
    m_nRefCount--;

    // If the reference count has got down to 0, delete us
    if ( m_nRefCount == 0 )
    {
        // Delete us (cast just to be safe, so that any non virtual destructor
        // is called correctly)
        delete (CActionData*)this;

        // WE MUST NOT REFERENCE ANY MEMBER VARIABLES FROM THIS POINT ON!!!
        // For this reason, we must simply return 0 here, rather than dropping
        // out of the if statement, since m_nRefCount references memory which
        // has been released (and would cause a protection fault).
        return 0;

    } // End if ref count == 0

    // Return the reference count
    return m_nRefCount;
}
```

Studying the above function code you will note that the first thing the function does is decrement its internal reference count by one. If the reference count is still greater than zero then nothing much happens; the function simply returns the new reference count of the object. However, if the reference count reaches zero, then the object deletes itself from memory using the **this** pointer.

It is perfectly valid for an object to delete itself as long as you do not try to access any of the member variables from that point on. When delete is called to release the object, the destructor of the object will be called and therefore all member variables will have been released from memory. In the above code, you can see that we are very careful that we do not simply execute the last line of the function in the delete case. That is because the last line returns the value of a member variable which would no longer exist at this point and would thus cause a general protection fault. Therefore, inside the deletion case code block we return a value of zero for the reference count.

So we can see that the application should never explicitly delete an instance of this class but should instead treat it like a proper COM object using AddRef and Release semantics.

CActionData::~CActionData

The destructor of a class is the last method to be called. With our object, it will only be called once the Release method has issued the 'delete this' instruction in response to its reference count hitting zero. This is where the various modes of our CActionData objects can clean up their memory allocations. In our current object, there is only one real mode ACTION_PLAYSOUND. The object assumes that the string containing the wave filename was allocated using the _tcsdup function and as such will free the memory for this string as its last order of business.

```
CActionData::~CActionData( )
{
    // Release any memory
    switch ( m_Action )
    {
        case ACTION_PLAYSOUND:
            // Release duplicated string
            if ( m_pString ) free( m_pString ); // Allocated by '_tcsdup';
            break;

    } // End Action Switch
}
```

CActionData::QueryInterface

This function is quite an exciting one for us to look at. We have discussed how COM objects always expose this method as a means for checking the supported interfaces of an object. Indeed our object will expose two interfaces: the IUnknown interface (which all COM objects must expose) and the CActionData interface (our derived class). The CActionData interface will be used by the callback key registration callback function to assign context data to its member variables (such as the wave filenames in our example). The IUnknown interface will be used by the actor when it needs to release the callback data and has no idea of what type the context data actually points to.

As you will see, much of the mysteries of this function are removed when we look at how it is implemented in this simple example. Really, it is nothing more than a function that safely casts a pointer to the object (its 'this' pointer) to an interface/class of a specific supported type. Of course, a more

complex COM object might contain multiple classes such that some of the interfaces it exposes may work internally with completely separate objects, but in our case, there is only one object (a CActionData object). Therefore, this function is only concerned with returning one of two supported interfaces that allow the application to call the methods supported by this object.

As we have seen in the past, the QueryInterface method of IUnknown expects to be passed two parameters. The first is the alias of the GUID. As we discussed in Chapter Two, every interface has a GUID -- a unique number that identifies it. We also discussed how each GUID will often also have an alias to make it easier to remember and use. These aliases are usually in the form of "IID_I***name***", where *name* is the name of the interface. For example, if we wish to enquire about an object's support for the ID3DXAnimationSet interface, we can simply specify that interface using the IID_ID3DXAnimationSet alias instead of using its harder to remember GUID. If we look in the D3DXAnim9.h header file for example, we will see that the GUID for this interface is assigned to the alias 'IID_ID3DXAnimationSet'. The header file or the manual that ships with a software component will usually be the place to obtain a list of all aliases/interface GUIDS supported by that installed component.

Because, we have added our own interface to the mix (CActionData), we have given this interface a unique GUID and have assigned it the alias IID_CActionData. If you look in CScene.h, you will see this alias defined as follows:

```
GUID IID_CActionData = {0x736FE02A, 0x717D, 0x491F, {0x87, 0xED, 0x4B, 0x16, 0x2, 0xC8, 0xF3, 0xCA}};
```

As you can see, IID_CActionData is a variable of type GUID. The values assigned to it (i.e., the numbers in the curly braces) are the GUID itself. You can generate unique GUIDS for your own classes/COM interfaces using the GuidGen.exe tool that ships with Microsoft Visual C++. The GUIDs generated are guaranteed to be unique with respect to the GUIDs of all other interfaces in existence, so our function can safely assume that when this GUID is passed, the caller is definitely asking whether the object supports our CActionData interface, which it does.

The second parameter passed into the function should be the address of a void pointer which on function return will contain the address of the requested interface, or NULL if the GUID passed in was for an interface type that the object does not support. The application can then cast the void pointer to the correct type.

As you can see by looking at the code below, it really is just a list of conditional statements that test the past GUID to see if it is of any of the types supported by the object. If it is, we cast the ***this*** pointer, a pointer to the C++ object itself, to the desired base class before casting again to the void pointer which is ultimately returned to the function.

```
HRESULT CActionData::QueryInterface( REFIID iid, LPVOID *ppv )
{
    // Double cast so compiler can decide
    // whether it needs to switch to an alternate VTable
    if ( iid == IID_IUnknown )

        *ppv = (LPVOID)((IUnknown*)this);
```

```
    else if ( iid == IID_CActionData )

        *ppv = (LPVOID)((CActionData*)this);

    else
    {
        // The specified interface is not supported!
        *ppv = NULL;
        return E_NOINTERFACE;

    } // End iid switch

    // Call addref since a pointer is being copied
    AddRef();

    // We're supported!
    return S_OK;
}
```

When this function returns, the caller will be passed a pointer (via parameter two) to the supported interface that they requested. Notice that this function must increment the reference count (AddRef) before the function returns, since we have issued a new interface pointer to the object and an additional pointer to this object is now in existence.

The only thing that may seem confusing about the above code is how we cast the object's pointer to the requested interface/base class type before casting it to a void pointer. As we are simply casting the object to a base class, why not just cast 'this' straight to a void pointer? In this particular situation we could do that, however the double cast ensures that our function will work correctly even if any of the interfaces are derived from multiple base classes (multiple inheritance). So we should practice this safe cast policy regardless. If for example, the CActionData interface was later revised such that it had more than one base class (IUnknown and some other base class), this would cause problems as the complier would have no idea which vtable to return to the caller when the object was cast straight to a void pointer. When a class is derived from multiple base classes, it actually has multiple vtable pointers stored in the member area of each instance. By casting the 'this' pointer to the correct interface type first, *before* casting it to void, we are allowing the compiler to determine which vtable should be used for accessing functions via this interface.

> **Note:** If you would like to understand more about implementing COM objects from C++ objects, you should check out the following excellent free tutorial. It takes you step by step through the process of turning a C++ object into a COM object. It discusses the topic of multiple inheritance and multiple vtables also. This tutorial is called 'From CPP to COM' and can be found at:
>
> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomg/html/msdn_cpptocom.asp

We have now implemented the three methods of IUnknown so that our object will obey the lifetime encapsulation rules set out by the COM specification. It also (as dictated by the COM standard) exposes a QueryInterface method that allows an application to ask our object if a particular interface is supported and have returned a pointer to that interface if it is. Our simple QueryInterface implementation is really just a function that allows the application to safely ask the question, "Can I cast this object pointer to a specific class type?"

We have now covered the first new class of this project (CActionData). In summary, it is a support class that allows us to store arbitrary data in an object that is capable of handling its own destruction. As discussed and demonstrated, we use an instance of this class to store the context data for each callback key that we register. In Lab Project 10.1, each callback key is used to play a sound, so the filename of the wave file is stored in CActionData object and assigned to the context data pointer of the D3DXKEY_CALLBACK structure contained in the actor's animation set. When the actor is being destroyed and the animation set is released, the actor will fetch every callback key and cast its context data pointer from a void pointer to an IUnknown interface. The actor can then call IUnknown::Release on this pointer, causing the CActionData object to destroy itself. The actor has no knowledge of what this pointer actually points to; only that it supports the Release method and will handle its own destruction. Therefore, the only rule that our actor class places on the callback system is that the context data pointer for a callback key must point to an IUnknown derived structure.

All of these pieces will fit together when we finally cover the CActor code. It was mentioned earlier that for the most part, the new methods we add to CActor should really only be lite wrappers around the underlying methods of the ID3DXAnimationController. After all, the ID3DXAnimationController handles most of the heavy lifting. We just have to assign animation sets to tracks on the mixer and play them back, right? Well, not quite! Things *should* be that simple, but unfortunately we will have to do some work just to get the animation system to work at all. This is due to a very nasty bug in ID3DXKeyframedAnimationSet.

Error in ID3DXKeyFramedAnimationSet::GetSRT

In the summer of 2003, the DirectX 9.0b SDK was released with a fully overhauled animation system. It is this new system that we have covered in this course. The changes to the animation system from previous versions were significant enough that code that had been written to work with previous versions would no longer compile.

The animation system was streamlined by the moving of animation data (keyframes) into the animation set's themselves. This had not previously been the case, as the animation data used to be stored in separate 'interpolator' objects which then had to be registered with an animation set. It is likely that this additional layer was disposed of to make the accessing of animation data by the animation set more efficient.

Previously, no AdvanceTime function existed either; in its place was a function called SetTime. Instead of passing in an elapsed time to this function, you had to pass in an actual global time which would in turn set the global time of the controller explicitly. This placed the burden on the application to keep track of the current global time of the animation controller by summing the elapsed time between frame updates. This burden has now been removed and the application now simply passes the elapsed time into the AdvanceTime call, allowing the controller to keep track of global time on our behalf.

In the 2003 summer update, the callback system was also added to the animation system. This is another worthy addition to the animation system, exposing better communication possibilities between an application and a currently playing animation.

While this animation system re-write undoubtedly caused some headaches for developers that had already coded their own animations using the previous system, the new animation system was arguably the better for the overhaul. However, in the re-write of the animation system, an error crept into the ID3DXKeyframedAnimationSet::GetSRT method, making the ID3DXKeyframedAnimationSet object essentially useless for serious development. This error seems to have gone unnoticed by the developers (and the beta testers) and as of the SDK summer update of 2004 (DirectX 9.0c) released a whole year later, the bug was still there.

The reason the bug has probably gone unnoticed for so long is that it does not always make itself apparent. For simple animated X files which play at a medium speed with basic animations, the GetSRT function seems to handle these cases fine and generates the correct SRT data for the periodic position of an animation. One such X file that seems to play without any problems using the new animation system is, ironically, Microsoft's tiny.x (ships with the SDK). This is probably the file that everyone uses to test the animation system, which might explain the oversight. Only when we began to use more complex X files did we discover the bug here in the Game Institute labs.

Indeed, the bug is actually quite hard to see if you are not paying very close attention to the screen, since the animations seem to play out correctly for the most part. However, it seems that at certain periodic positions in some animations, a single GetSRT function call will generate wildly incorrect data. This seems to happen almost at random. As the following call to GetSRT will once again generate the correct SRT data, this causes small but noticeable glitches in the animation being played. Basically, for only a few milliseconds, objects in your scene will skip or jump to completely incorrect positions in the scene as their matrices are built from bogus SRT data. A few milliseconds later, the objects are back in their correct position until the next time the GetSRT function generates invalid data. If you play the animation set from Lab Project 10.1 using the vanilla D3DX animation system, you will see these glitches as the animation plays out.

So what are the implications for the D3DX Animation System?

Since D3DXLoadMeshHierarchyFromX will always load AnimationSet objects defined in X files into ID3DXKeyframedAnimationSets, you might assume that the entire D3DX animation system that we have spent time learning about is now useless. But this is certainly not the case. We discussed in the textbook that both ID3DXKeyframedAnimationSet and D3DXCompressedAnimationSet are derived from the base interface ID3DXAnimationSet. We also learned that how we have the ability to derive our own classes from ID3DXAnimationSet so that we can create our own custom animation sets and plug them into the D3DX animation system. Provided our derived class implements the functions of the base class (GetSRT, GetPeriodicPosition, etc.) the animation controller will work fine with our derived classes. So in this workbook, we are going to write our own keyframed animation set and use that instead. Essentially we will replace all ID3DXKeyframedAnimationSets (created and registered with the controller when we load an X file), with our own CAnimationSet objects after the X file has been loaded.

On the bright side, working around this bug presents us with an opportunity to really get under the hood of the animation set object and learn how the keyframe interpolation and callback key searching is performed. Remember, it is the animation set that is responsible for generating the SRT data from its

keyframes and for returning the next scheduled callback key to the controller. We will have to implement all of this in our own animation set class, which will give us quite a bit of new insight.

> **Note:** Hopefully, by the time you read this, Microsoft will have fixed this bug, making our workaround unnecessary. However, this section will teach you how animation sets work as well as go a long way towards explaining how to create your own COM objects. It is therefore recommended that you read this section, even if the bug no longer exists. It will be helpful if you need to expand the D3DX animation system (or even write your own system from scratch) for your own reasons down the road.

Although we will not implement our animation set class as a true COM object (e.g., it will not be registered with the Windows registry), it will be close enough to the real thing. As such, we will need to implement the three methods from IUnknown (AddRef, Release, and QueryInterface), just like we did with our CActionData class.

Replacing the ID3DXKeyframedAnimationSets that have been loaded and registered with the animation controller (by D3DXLoadMeshHierarchyFromX) with our own CAnimationSet objects will be a trivial task. The CActor will have a function called ApplyCustomSets which will only be called by the actor after the X file has been loaded. The task of this function will be to copy the animation data (i.e., the keyframes) from each registered ID3DXKeyframedAnimationSet into a new CAnimationSet object. Once we have copied all the keyframed animation sets into their CAnimationSet counterparts, we will un-register each original keyframed animation set from the controller and release it from memory. We will then register our newly created CAnimationSet copies with the controller to take their place. The CAnimationSets will contain the same keyframe data and callback keys as the original ID3DXKeyframedAnimationSets from which they were copied. What we have essentially done is *clone* the animation information from each keyframed animation set into our own CAnimationSet objects. At the end of this process, our animation controller will now have *n* CAnimationSets registered with it instead of *n* ID3DXKeyframedAnimationSets. We will see how this works shortly.

The next class we will write in this lab project will be the CAnimationSet since it will be used by the code we add to CActor. It should also still be fresh in your mind after reading the textbook exactly what interaction between the animation controller and the animation set takes place. Certainly our class will need to answer requests from the animation controller in the proper way. It will need to be able to return the SRT data for a given periodic position (when the controller calls GetSRT for one of its animations) and it will also need to be able to search for the next scheduled callback key when one is requested by the controller, making sure that any passed flags (such as the one to ignore the initial position in the search) are correctly adhered to.

Source Code Walkthrough - CAnimationSet

The class declaration for CAnimationSet (see below) can be found in CActor.h and its implementation is in CActor.cpp. Notice how we derive our class from the ID3DXAnimationSet abstract base class. We did a similar thing in the last chapter when we derived our own class from ID3DXAllocateHierarchy. By doing this, we can plug our animation set object into the D3DX animation system and the controller will treat it as an ID3DXAnimationSet interface. Looking at the member functions, we can see that we must implement the three IUnknown methods (QueryInterface, AddRef, and Release) so that our object

follows the required lifetime encapsulation for COM compliance. We must also implement all of the methods from the ID3DXAnimationSet base interface because the animation controller will expect to them be implemented when asking our animation set for SRT data and callback key information.

```
class CAnimationSet: public ID3DXAnimationSet
{
  public:
     CAnimationSet( ID3DXAnimationSet * pAnimSet );
    ~CAnimationSet( );

    // IUnknown
    STDMETHOD(QueryInterface)(THIS_ REFIID iid, LPVOID *ppv);
    STDMETHOD_(ULONG, AddRef)(THIS);
    STDMETHOD_(ULONG, Release)(THIS);

    // Name
    STDMETHOD_(LPCSTR, GetName)(THIS);

    // Period
    STDMETHOD_(DOUBLE, GetPeriod)(THIS);
    STDMETHOD_(DOUBLE, GetPeriodicPosition)(THIS_ DOUBLE Position);

    // Animation names
    STDMETHOD_(UINT, GetNumAnimations)(THIS);
    STDMETHOD(GetAnimationNameByIndex)(THIS_ UINT Index, LPCTSTR *ppName);
    STDMETHOD(GetAnimationIndexByName)(THIS_ LPCTSTR pName, UINT *pIndex);

    // SRT
    STDMETHOD(GetSRT)(THIS_
                          DOUBLE PeriodicPosition,   // Position in animation
                          UINT Animation,            // Animation index
                          D3DXVECTOR3 *pScale,       // Returns the scale
                          D3DXQUATERNION *pRotation, // Returns the rotation
                          D3DXVECTOR3 *pTranslation);// Returns the translation

    // Callbacks
    STDMETHOD(GetCallback)(THIS_
                           DOUBLE Position, // Position to start callback search
                           DWORD Flags,               // Callback search flags
                           DOUBLE *pCallbackPosition, // Next callback position
                           LPVOID *ppCallbackData);   // Returns the callback data

  private:
    ULONG               m_nRefCount;      // Reference counter
    LPTSTR              m_strName;        // The name of this animation set
    CAnimationItem     *m_pAnimations;    // An Array of animations
    ULONG               m_nAnimCount;     // Number of animation items.
    D3DXKEY_CALLBACK   *m_pCallbacks;     // List of all callbacks stored
    ULONG               m_nCallbackCount; // Number of callbacks in the above list
    double              m_fLength;        // Total length of this animation set
    double              m_fTicksPerSecond;// The number of source ticks per second
};
```

Let us first examine the class member variables of this class since they will provide a better insight into how we will implement our functionality.

ULONG m_nRefCount
This member will contain the object reference count. When the object is first created, this value will be set to 1 in its constructor. If the application makes a copy of a pointer to the CAnimationSet it should call the CAnimationSet::AddRef method to force the reference count to be incremented. This will make sure that the reference count of our object is always be equal to the number of pointer variables currently pointing to it that have not yet had Release called. The Release method of this class will simply decrease the reference count and cause the object to delete itself from memory if the reference count reaches zero.

When we register the animation set with the D3DX animation controller, the controller will also call the AddRef function of our animation set object and increase the reference count. This is because, until the animation set is un-registered from the controller or the controller is released, the reference count of our object should reflect that the controller has stored a copy of the pointer. Even if our application has called Release on all of its outstanding pointers to a given animation set, the animation set would not be released from memory until the controller no longer needed access to it. In this example, only when the controller was destroyed or the animation set was un-registered, would the controller call the Release method (causing our reference count to hit zero and the object to be deleted from memory).

LPTSTR m_strName
This member is a string that will contain the name of the animation set. In our constructor, this will be extracted from the ID3DXKeyframedAnimationSet that our object is going to replace.

D3DXKEY_CALLBACK *m_pCallbacks
This variable will contain any callback keys that existed in the original ID3DXKeyframedAnimationSet passed into the constructor. The constructor will fetch all the callback keys from the original animation set and allocate this array to store them.

ULONG m_nCallbackCount
This member contains the number of callback keys in the m_pCallbacks array. If this value is set to zero then the animation set has no callback keys defined.

double m_fLength
This member will be set in the constructor to the value of the period of the original ID3DXKeyframedAnimationSet that is being copied. Therefore, this value contains the length of the animation set. We know that this is also the length of the longest running animation in the set (i.e., the highest value timestamp of all keyframes in all animations).

It is this value that will be returned to the animation controller when it calls our CAnimationSet::GetPeriod method. Our animation set also uses this value to calculate a periodic position. If looping is enabled for this animation set, then the periodic position will always be the product of an fmod between the position of the track (passed in by the controller) and the period (length) of the animation set.

double m_fTicksPerSecond
As discussed in the textbook, keyframes do not define their timestamps in seconds but rather using an arbitrary timing system called *ticks*. As our animation set's GetSRT method will always be passed a time value in seconds by the controller, it must be able to convert the seconds value into ticks so that it

can be used to find the two bounding keyframes in each of the S, R and T arrays. This value, which will be copied over in the constructor from the original keyframed animation set, specifies how many of these ticks represent a second. We can convert a value in seconds into a tick value by multiplying by m_fTicksPerSecond.

Callback keys are also defined in ticks, so this member will also be needed to convert between seconds and ticks in the GetCallback method of our class. You will recall that this function is called by the controller to locate the next scheduled callback event. In that case, the controller passes in a current position (in seconds) to begin the search. Using this value, we can convert the initial position and the callback key timestamps into the same timing system for the comparison. GetCallback will return the timestamp of the next scheduled callback key to the controller in seconds not in ticks.

CAnimationItem *m_pAnimations

In the textbook we learned that each animation set contains a number of animations and that an animation is really just a container of keyframe data for a single frame in the hierarchy. If an animation set animates five frames in a hierarchy, it will have five animations defined for it. The name of the frame in the hierarchy will match the name of the animation in the animation set that contains the keyframe data to animate it. This is how the animation controller recognizes the the pairing of hierarchy frames and animations when the set is registered with the controller.

In our implementation of CAnimationSet, we have decided to represent each animation contained in the set as a CAnimationItem object. This is another new class we will introduce in this project. If the animation set has ten animations, then this pointer will point to an array of ten CAnimationItem objects. We will cover the code to the CAnimationItem object after we have discussed CAnimationSet. We can think of a single CAnimationItem object as containing the data of a single Animation data object in an X file.

So CAnimationSet is essentially a CAnimationItem manger. The CAnimationItem object will store the keyframes (SRT Arrays) for a single frame in the hierarchy and be responsible for the keyframe interpolation (via its GetSRT method). The CAnimationSet::GetSRT function will really just act as a wrapper. Its job will be to locate the CAnimationItem that the controller has requested SRT data for, and call its CAnimationItem::GetSRT function. This function will return the SRT data back to the animation set which will in turn return it back to the controller.

ULONG m_nAnimCount

This final member stores the number of animations in the animation set (i.e., the number of elements in the array pointed to by m_pAnimations).

CAnimationSet::CAnimationSet

The constructor takes a single parameter, a pointer to an ID3DXAnimationSet interface. It is this animation set that will have its data extracted and copied by the constructor. We have previously discussed that our CActor will be calling this constructor and will pass in an ID3DXKeyframedAnimationSet so that we can generate a replacement CAnimationSet object which corrects the bug in the GetSRT method. Therefore, this constructor will need to extract the keyframe

data from the passed animation set so that it can make its own copy. However, the base ID3DXAnimationSet interface does not support functions for retrieving the keyframe data so we will first need to query the passed object for an ID3DXKeyframedAnimationSet interface. If the function succeeds then we will have an ID3DXKeyframedAnimationSet interface from which we can extract the data. If the QueryInterface method fails, then the interface we have been passed is attached to an unsupported object which does not expose the ID3DXKeyframedAnimationSet interface. If this is the case, we throw an exception.

```
CAnimationSet::CAnimationSet( ID3DXAnimationSet * pAnimSet )
{
    ULONG   i;
    HRESULT hRet;
    ID3DXKeyframedAnimationSet * pKeySet = NULL;

    // Query to ensure we have a keyframed animation set
    hRet = pAnimSet->QueryInterface( IID_ID3DXKeyframedAnimationSet,
                                     (LPVOID*)&pKeySet );
    if ( FAILED(hRet) || !pKeySet ) throw hRet;
```

If we reach this point, then it means we have a valid interface to a keyframed animation set. The first thing we do is call the ID3DXKeyframedAnimationSet::GetName method to get the name of the keyframed animation set that we are copying. If the animation set has a name, then we use the _tcsdup string duplication function to make a copy of the string and store it in the_strName member variable. We also use the methods of ID3DXKeyframedAnimationSet to copy the period of the animation set and the ticks per second value into member variables of the of the CAnimationSet object.

```
    // Duplicate the name if available
    LPCTSTR strName = pKeySet->GetName( );
    m_strName = (strName) ? _tcsdup( strName ) : NULL;

    // Store total length and ticks per second
    m_fLength           = pKeySet->GetPeriod();
    m_fTicksPerSecond   = pKeySet->GetSourceTicksPerSecond( );
```

Our next task will be to copy over any callback keys that may be contained in the set we are copying. First we call the ID3DXKeyframedAnimationSet::GetNumCallbackKeys and, provided this is greater than zero, allocate the CAnimationSet's callback key array to store this many elements. We then call the ID3DXKeyframedAnimationSet::GetCallbackKeys method and pass in a pointer to this member array. This function will copy all the callback keys in the original animation set into our object's callback keys array.

```
    // Copy callback keys
    m_pCallbacks     = NULL;
    m_nCallbackCount = pKeySet->GetNumCallbackKeys( );
    if ( m_nCallbackCount > 0 )
    {
        // Allocate memory to hold the callback keys
        m_pCallbacks = new D3DXKEY_CALLBACK[ m_nCallbackCount ];
        if ( !m_pCallbacks ) throw E_OUTOFMEMORY;
```

```
        // Copy the callback keys over
        hRet = pKeySet->GetCallbackKeys( m_pCallbacks );
        if ( FAILED(hRet) ) throw hRet;

    } // End if any callback keys
```

The next task will be to find out how many animations exist inside the passed animation set so that we can allocate that many CAnimationItem objects. Remember, a CAnimationItem represents the data for a single animation. Therefore we call the ID3DXKeyframedAnimationSet::GetNumAnimations method and use the returned value to allocate the object's CAnimationItem array to hold this many elements. We then loop through each newly created CAnimationItem in the array and call its BuildItem item method. This method will be discussed when we cover the CAnimationItem code a little later, but as you can see in the following code snippet, we pass in the ID3DXKeyframedAnimationSet interface and the index of the animation we wish to have copied into our CAnimationItem. The CAnimationItem::BuildItem method is a simple function that will fetch the SRT keys for the passed animation and make a copy of them for internal storage. After a call to BuildItem, a CAnimationItem object will contain all the keyframes that existed in the passed animation set for the specified animation index. For example, if we call:

```
m_pAnimations[5].BuildItem( pKeySet , 5 )
```

then the sixth CAnimationItem in our array will contain the SRT keyframes from the sixth animation in the keyframed animation set (pKeySet).

Here we see the code that allocates our CAnimationItem array and populates each item with data from the source animation set:

```
    // Allocate memory for animation items
    m_pAnimations = NULL;
    m_nAnimCount  = pKeySet->GetNumAnimations();
    if ( m_nAnimCount )
    {
        // Allocate memory to hold animation items
        m_pAnimations = new CAnimationItem[ m_nAnimCount ];
        if ( !m_pAnimations ) throw E_OUTOFMEMORY;

        // Build the animation items
        for ( i = 0; i < m_nAnimCount; ++i )
        {
            hRet = m_pAnimations[ i ].BuildItem( pKeySet, i );
            if ( FAILED(hRet) ) throw hRet;

        } // Next animation

    } // End if any animations

    // Release our keyframed friend
    pKeySet->Release();
```

In the code above, you can see that after we have copied the animation data into each of our CAnimationItems, the ID3DXKeyframedAnimationSet interface that we queried for earlier is released.

This is important because every time we call QueryInterface for a COM object, the reference count is incremented just before the requested interface is returned. If we forgot to do this step then the animation set would not be released when the CActor decided it was no longer needed. Remember, after this constructor returns program flow back to CActor, the keyframed animation set that was passed into the function will no longer be needed and it will have its interface released and unregistered from the controller. This will drop the reference count of the original keyframed animation set to zero, causing it to unload itself from memory.

Finally, because we wish our CAnimationSet object to behave like a COM object, we must remember to set the initial reference count of the object to 1 in the constructor.

```
    // ************************************
    // *** VERY IMPORTANT
    // ************************************
    // Set our initial reference count to 1.
    m_nRefCount = 1;
}
```

Once the constructor exits, our object will have been created and populated with the information that was contained in the ID3DXKeyframedAnimationSet passed in by the actor. The reference count of our object will also be 1, which is exactly as it should be. You will see later when we cover the CActor code that this new CAnimationSet will then be registered with the controller, in place of the original.

CAnimationSet::~CAnimationSet

The destructor to this class is nothing special; it just releases any internal memory being used by the animation set. This will be the two arrays that contain the callback keys and the CAnimationItems as well as the memory that was allocated to store the name of the animation set when _tcsdup was called in the constructor. Remember that _tcsdup uses a C alloc style function for its memory allocation, so we must release the memory using the free function.

```
CAnimationSet::~CAnimationSet( )
{
    // Release any memory
    if ( m_pCallbacks  ) delete []m_pCallbacks; m_pCallbacks = NULL;
    if ( m_pAnimations ) delete []m_pAnimations; m_pAnimations = NULL;
    if ( m_strName     ) free( m_strName ); m_strName = NULL;
}
```

What is unique about the destructor for this class is that the application code should never directly invoke it. So we should never ever write code like the following:

```
delete pMyAnimSet;
```

Our object is going to behave like a proper COM object and we never directly delete a COM object. Instead we call the interface's Release method which decrements the reference count of the object. When the reference count drops to zero, the object should delete itself from memory by doing:

```
delete this;
```

This is exactly what happens in our Release method (see next). The destructor is only ever invoked from within the class Release method.

CAnimationSet::Release

When the animation controller is destroyed by the actor, the controller will call Release for each of its registered animation sets. Therefore, our animation set must properly handle this call in COM fashion. This code is almost identical to the Release method of our CActionData class, which also followed the COM protocol.

Release decrements the reference count. When the count reaches zero, the object deletes itself by making the 'delete this' call. When we do a 'delete this', the object destroys itself in the standard way, and the destructor is invoked. When program flow finally returns back to the Release method, the object and all its member variables will have been destroyed. We must make sure that this method never tries to access any member variables or functions after this deletion (even the *this* pointer no longer exists). This is like being stranded in a static function which has no instance associated with it. As you can see, we simply return 0 after the delete.

```
ULONG CAnimationSet::Release( )
{
    // Decrement ref count
    m_nRefCount--;

    // If the reference count has got down to 0, delete us
    if ( m_nRefCount == 0 )
    {
        // Delete us (cast just to be safe, so that any non virtual destructor
        // is called correctly)
        delete (CAnimationSet*)this;

        // WE MUST NOT REFERENCE ANY MEMBER VARIABLES FROM THIS POINT ON!!!
        // For this reason, we must simply return 0 here, rather than dropping
        // out of the if statement, since m_nRefCount references memory which
        // has been released (and would cause a protection fault).
        return 0;

    } // End if ref count == 0

    // Return the reference count
    return m_nRefCount;
}
```

Note that if the object is not actually destroyed, we just decrement the reference count and return the updated value.

CAnimationSet::AddRef

When the actor adds a CAnimationSet object to the animation controller, the animation controller will expect this function to be implemented. This is because it will call the AddRef method when it stores a copy of the animation set's pointer. As you can see, this is a simple one line function that just increments the reference count.

```
ULONG CAnimationSet::AddRef( )
{
    return m_nRefCount++;
}
```

CAnimationSet::QueryInterface

The final IUnknown method we must implement is now familiar to us, given our coverage of the CActionData class. The QueryInterface function is passed an interface GUID and a void pointer and if the passed GUID interface is supported by this object, then on function return the passed void pointer will point to a properly cast version of the object.

If you take a look in CActor.h you will see that the GUID we generated (via GuidGen.exe) for the CAnimationSet interface is defined as follows, along with an IID_CAnimationSet alias:

```
GUID IID_CAnimationSet = {0x5B23B100, 0xE885, 0x4F63, {0xB0, 0x2E, 0x50, 0xFC, 0x99, 0xBA, 0x3, 0xDD}};
```

We will support three interfaces for this object. First, we obviously want to support the top level CAnimationSet interface since this is the actual type of the object. We will also support the casting of our object to an ID3DXAnimationSet interface. Remember, this interface is the base interface for all animation sets that are compatible with the D3DX animation controller. Regardless of the type of animation set registered with a controller (custom or otherwise), the controller will see all animation sets as being of type ID3DXAnimationSet. Therefore, every animation set we register must support this interface. This means of course that all the methods of this interface must be supported also. Our CAnimationSet is derived from ID3DXAnimationSet so we should implement all the methods of this interface in our object also. The third and final interface we must implement functions for is IUnknown. ID3DXAnimationSet is derived from this interface and since our class is derived from ID3DXAnimationSet, we must support the methods of its base class also. In short, the three interfaces we support are IUnknown, ID3DXAnimationSet, and CAnimationSet. Therefore, our QueryInterface implementation should allow the casting of the 'this' pointer to any of these three types.

```
HRESULT CAnimationSet::QueryInterface( REFIID iid, LPVOID *ppv )
{
    // We support three interfaces, the base ID3DXAnimationSet, IUnknown
     // and ourselves.
    // It's important that we cast these items, so that the compiler knows we
    // possibly need to switch to an alternate VTable
    if ( iid == IID_IUnknown )
        *ppv = (LPVOID)((IUnknown*)this);
```

```
    else if ( iid == IID_ID3DXAnimationSet )
        *ppv = (LPVOID)((ID3DXAnimationSet*)this);

    else if ( iid == IID_CAnimationSet )
        *ppv = (LPVOID)((CAnimationSet*)this);

    else
    {
            // The specified interface is not supported!
            *ppv = NULL;
            return E_NOINTERFACE;

    } // End iid switch

    // We're supported!
    AddRef();
    return S_OK;
}
```

Notice the double cast of the 'this' pointer – something we did in the CActionData::QueryInterface method previously. Remember, this aids the compiler in choosing the correct vtable pointer to return if multiple base classes are being derived from.

Having implemented the required IUnknown functions, we must now implement the methods of the other interface from which this object is derived: ID3DXAnimationSet. All such methods must be implemented since they will be used by the animation controller when registering and playing back the animation set. Remember, ID3DXAnimationSet is a pure abstract base class, so all of its functions *must* be implemented in our derived class.

CAnimationSet::GetName

This function is called by the animation controller to fetch the name of an animation set. We simply return a pointer to the string member variable containing the name of the animation set. The name of our animation set was copied from the ID3DXKeyframedAnimationSet passed into the constructor.

```
LPCTSTR CAnimationSet::GetName( )
{
    return m_strName;
}
```

CAnimationSet::GetPeriod

The GetPeriod method should return the length of the animation set (copied from the ID3DXKeyframedAnimationSet interface passed into the constructor). The period of an animation set is defined by its longest running animation. The longest animation is the animation which has the keyframe with the highest timestamp in the set. The period of the animation is the timestamp of this keyframe converted from ticks into seconds.

```
DOUBLE CAnimationSet::GetPeriod( )
{
    return m_fLength;
}
```

If the keyframe with the highest timestamp value had a timestamp of 2400 ticks, the period would be calculated as 2400 * TicksPerSecond. Since this value never changes, it is calculated once when the animation set is first created and then cached. As we are copying the information from an already created ID3DXKeyframedAnimationSet, we can simply extract this value and copy it into our member m_fLength variable. Refer back to our coverage of the constructor to see this value copied from the passed keyframed animation set.

CAnimationSet::GetPeriodicPosition

We discussed the duties of this function many times throughout the textbook since it is one of the most important in the system. GetPeriodicPosition will be called by the animation controller and is passed a position value in seconds. This position is actually the current time of the track to which the animation set is applied. As this position may be well outside the total length of the animation set, this function is responsible for returning a periodic position that represents the behavior of the animation set.

As an example, if the animation set was configured to loop, then the position should be mapped such that the track position should always be converted into the timeframe of the animation. If the period of the animation set was 20 seconds and a track position of 110 seconds was passed in, this would be mapped to a periodic position of 10 seconds (10 seconds into its 6^{th} loop).

Obviously, if the animation set was configured not to loop, then this would return a periodic position of 20 seconds, the last keyframe in the animation set. That is, after the track position passes 20 seconds, the same periodic position for all subsequent track positions would be returned and the same SRT data eventually generated. In this case, it would appear as if the animation set stopped playing after 20 seconds.

In our class we will treat all animation sets as looping animations. Therefore, to map the passed track position to a periodic position (a local animation set time in seconds) we simply perform the modulus of the passed track position and the period (length) of the entire animation set.

```
DOUBLE CAnimationSet::GetPeriodicPosition( DOUBLE Position )
{
    // To map, we'll just loop the position round
    double Pos = fmod( Position, m_fLength );
    if ( Pos < 0 ) Pos += m_fLength;
    return Pos;
}
```

> **Note**: In the GetPeriodicPosition code we are careful to handle the case where the position may be specified as a negative value. For example, if the position was set to -12 for a 10 second animation then this should be treated as a 2 second loop around from the beginning of the animation. If the period of the animation was 10 seconds then -12 mod 10 would be -2. If we add the period of the animation to this (10 seconds) we get 8 seconds.

You are encouraged to add additional mapping modes beyond the simple looping logic we have implemented here. You could implement ping-pong mapping or return a periodic position that is clamped to the period of the animation set. You could also implement your own custom periodic mapping modes as well.

The final periodic position returned from this function is defined in seconds and is used by the animation controller to pass into the animation set's GetSRT method (once for each of its animations). The periodic position will be used to find the two keyframes from each SRT list that bound the time. Once the keyframes that bound the periodic position are found, they can be interpolated using a function of time. We will see this implementation in a moment when we examine the GetSRT method.

CAnimationSet::NumAnimations

This method is declared in the ID3DXAnimationSet interface and is used by the animation controller to determine how many animations are contained in the animation set. The controller must have this information so that it can set up the loop that iterates through every animation in an animation set. For each animation, it calls the GetSRT method to generate its SRT data for the current periodic position. Basically, the value returned from this method tells the controller how many GetSRT calls must be made to update this animation set.

```
UINT CAnimationSet::GetNumAnimations( )
{
    return m_nAnimCount;
}
```

This value was originally extracted from the keyframed animation set passed into the constructor. It describes the number of frames in the hierarchy that are being animated by this animation set.

CAnimationSet::GetAnimationNameByIndex

This method is called by the controller to fetch the name of a given animation in the animation set. It can be used to determine which frame in the hierarchy is being animated by the animation. This works because the name of the animation will always be the same as the name of the frame in the hierarchy which it animates.

```
HRESULT CAnimationSet::GetAnimationNameByIndex ( UINT Index, LPCTSTR *ppName )
{
    // Validate
    if ( !ppName || !*ppName || Index >= m_nAnimCount ) return D3DERR_INVALIDCALL;

    // Copy the selected item's name (it must always exist).
    _tcscpy( (LPTSTR)*ppName, m_pAnimations[ Index ].GetName() );

    return D3D_OK;
}
```

The function will be passed two parameters, the index of the animation in the set that is being inquired about and the address of a string pointer. The function uses the passed index to access its array of animations. (Remember, each animation in our array is actually a CAnimationItem object.) We copy the name of the animation at the specified index using the _tcscpy function, which we know will allocate memory for the string prior to copying it. We then assign the string pointer passed in to point to the name we have just duplicated. When the function returns, the name of the animation at the requested index (assuming it is a valid index) will be pointed to by the pointer passed in.

CAnimationSet::GetAnimationIndexByName

This next method (also from ID3DXAnimationSet) is used by the controller to fetch the index of the animation associated with the passed name. When an animation set is being registered, the animation controller calls the animation set's GetAnimationIndexByName function for each frame in the hierarchy. The controller expects to get back either a valid index via the pIndex output parameter and D3D_OK, or D3DERR_NOTFOUND via the function result.

This function allows the animation indices to be cached by the animation controller, removing the need to look up the index (or name) each time. It also helps the controller know which frames in the hierarchy even have animation data (those which resulted in D3DERR_NOTFOUND obviously do not having a matching CAnimationItem).

Once the animation controller caches the index of each animation in the set, it calls the animations set's AddRef method to increase its reference count. At this point the animation set is considered to be registered with the controller.

As you can probably guess, this is a function that simply loops through each of its animations, fetches the name of each animation and performs a string compare between the animation name and the string passed into the function. If a match is found, the index of the animation is copied into the integer whose address was passed in as the second parameter and the function returns success.

```
HRESULT CAnimationSet::GetAnimationIndexByName ( LPCTSTR pName, UINT *pIndex )
{
    ULONG i;

    // Validate
    if ( !pIndex || !pName ) return D3DERR_INVALIDCALL;

    // Search through our animation sets
    for ( i = 0; i < m_nAnimCount; ++i )
    {
        // Does the name match ?
        if ( _tcsicmp( pName, m_pAnimations[i].GetName() ) == 0 )
        {
            // We found it, store the index and return
            *pIndex = i;
            return D3D_OK;

        } // End if names match
```

```
    } // Next Item

    // We didn't find this item!
    return D3DERR_NOTFOUND;
}
```

If no match is found, we return D3DERR_NOTFOUND to the caller.

CAnimationSet::GetSRT

We have discussed at length the importance of the animation set's GetSRT function and the role it plays in the overall animation system. It is probably not overstating the point to say that this method is the core of the entire animation engine.

GetSRT is called by the controller for each animation in the animation set. Its job is to calculate the SRT data for the current periodic position. The periodic position is passed into the function along with the index of the animation that we wish to generate SRT data for. The method is also passed a scale vector, a rotation quaternion, and a translation vector, which on function return should be filled with the SRT data for the specified time.

While you might expect this function to be large, it is actually just a simple wrapper around the CAnimationItem::GetSRT method, which we will discuss shortly.

```
HRESULT CAnimationSet::GetSRT( DOUBLE PeriodicPosition, UINT Animation,
                               D3DXVECTOR3 *pScale, D3DXQUATERNION *pRotation,
                               D3DXVECTOR3 *pTranslation )
{
    // Validate
    if ( Animation > m_nAnimCount ) return D3DERR_INVALIDCALL;

    // Return the interpolated result
    return m_pAnimations[ Animation ].GetSRT( PeriodicPosition,
                                              pScale,
                                              pRotation,
                                              pTranslation );
}
```

As you can see, the function uses the passed animation index to access the CAnimationItem in its animation array. It then calls its GetSRT method passing in the periodic position and the three structures that will contain the SRT output. Remember, the CAnimationItem class represents the data and members for a single animation in the set. It is a keyframe container that exposes its own GetSRT method which performs the interpolation to generate the SRT data for the passed periodic position for a single animation.

If the animation set contained 50 animations, then we would expect the animation controller to call the CAnimationSet::GetSRT method 50 times in a single update. Each time, the index of a different animation would be passed. This is used to call the GetSRT method of the corresponding CAnimationItem in the array and generate its SRT data. Our function is really nothing more than a function that routes an animation request to the GetSRT method of the correct animation.

CAnimationSet::GetCallback

The final method that we need to implement in our CAnimationSet class is called GetCallback (an ID3DXAnimationSet interface method). As discussed in the textbook, this method is passed a track position by the controller and will perform a search starting from that position to locate the time and data for the next scheduled callback event in the track timeline. Remember, the controller will cache this callback information and will not need to call this method again until the callback has been executed.

The code that follows shows the first section of the function. The function is passed a time (in seconds) by the animation controller describing the track position from which to begin the search for the next callback key. The time value passed in depends on how the function is being called by the animation controller. As explained in the textbook, if the controller is having AdvanceTime called for the first time, or if the global time of the controller has been reset, the callback event cache of the controller will be in an invalid state. When this is the case, the controller will call the GetCallback method of the animation set passing in the current position of the track (i.e., the track time). It will be returned the next scheduled callback event on the track timeline at some point in the future. The animation controller will then cache this information so that it knows it does not have to perform this search again until the position of the track has passed the cached callback key timestamp causing event execution.

If the callback event cache is not invalid, then the controller will call this function passing in the previously cached event time as the position parameter. For example, imagine the first time the AdvanceTime method is called and the controller calls the animation set's GetCallback method, passing in the current track position (0 in this case). The function might return a callback key with a timestamp of 10 seconds. This information will be cached by the controller so that it knows it does not have to do anything else with the callback system until 10 seconds (track time) has been reached. Approximately 10 seconds later, when the AdvanceTime call happens again, the controller would determine that the current track position is now larger than the previously cached callback key timestamp. The callback key is executed (i.e., the callback handler is called) and now it is time to find a new 'next' scheduled callback key. The controller calls the GetCallback method of the animation set again but will not pass in the current track position (which may be larger than 10 seconds). Instead, the previously cached callback time (10 seconds) will be passed in as the position parameter. This will cause the function to begin searching for a new callback key from the position of the previous callback key.

Remember that if the above system was not employed, callback events could be skipped. For example, imagine that we have a scheduled next callback event of 10 seconds, but the next time GetCallback is called, 12 seconds have passed. The 10 second event would be triggered normally because the controller would correctly detect that the position of the track (12 seconds) has passed the currently cached callback time (10 seconds). However, if we were to then perform the search for the next scheduled callback key starting at the current track position (12 seconds), we have risked skipping an event that may have been registered at 11 seconds because we jumped straight from 10 to 12 seconds. Therefore, by always searching for the next scheduled callback event from the position of the previously cached callback key, we make sure that we catch and trigger all callback events, even if the actual track position is several seconds ahead of the callback key processing pipeline.

The second parameter to this function can be a combination of zero or more flags that control how the search is performed.

If the controller passes in the D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION flag, then our function should not return a callback key which shares the same timestamp as the initial position passed in. In the textbook we learned that the controller will always pass this flag if a previously cached callback exists. This is because the position passed in will be the time of the previous timestamp and, if we do not ignore the initial position in the search, we will end up returning that same callback key that has already been executed. The exception is when the controller has an invalid cache; in this case, no previous callback exists and the controller is interested in the very first callback key that exists from the passed initial position (including that initial position).

Another flag that can also be combined in this parameter is the D3DXCALLBACK_SEARCH_BEHIND_INITIAL_POSITION flag, which specifies that the call would like the search for the next callback key to be done in the reverse direction (i.e., searching backwards from the passed initial position to find the next 'previous' key). Interestingly, in our laboratory tests, we never seemed to be passed this flag by the D3DXAnimationController, so it is very hard to determine with accuracy when the controller might pass this flag. It may be that it never does and that this is simply a flag that the application can use for its own callback key queries. You might assume that it is used by the controller when an animation set is playing in ping-pong mode and the direction of the animation changes with every odd and even loop. However, this is highly unlikely since the GetPlayBack type method is not a member of the ID3DXAnimationSet base interface and therefore the controller would have no way of accessing whether an animation set was configured to loop or ping-pong. At the time of writing there was no exporting software supporting the ping-pong animation mode, so these are ultimately untested conjectures. So it may be that this search mode is just for an application's benefit. After all, use of the GetCallback method is not restricted to the controller; it can also be a utility function used by your application to search for callback keys. Either way, the ID3DXKeyframedAnimationSet implementation supports the backwards search mode, so our implementation will do so as well.

The final two parameters are output parameters. The controller will pass in the address of a double and the address of a void pointer and on function return, the double should contain the track time (in seconds) of the next scheduled callback key and the void pointer should point to any context data (a CActionData object in our case) that was associated with the callback key when it was registered. This context data will be passed to the callback handler function (covered later) when the callback event is executed.

Here is the first small section of the function that requires some explanation:

```
HRESULT CAnimationSet::GetCallback( DOUBLE Position, DWORD Flags,
                                    DOUBLE *pCallbackPosition,
                                    LPVOID *ppCallbackData )
{
    long                i;   // Must be signed long for the backwards search
    D3DXKEY_CALLBACK    Key;

    // Are there any keys at all?
    if ( m_nCallbackCount == 0 ) return D3DERR_NOTFOUND;

    // Modulate position back into our range (and convert to ticks)
    double fPeriodic = fmod( Position, m_fLength ) * m_fTicksPerSecond;
    double fLoopTime = (Position * m_fTicksPerSecond) - fPeriodic;
```

The first thing the code does is test to see if this animation set even has any callback keys registered. If not, the function can return immediately. Notice that we return the D3DERR_NOTFOUND result expected by the controller when no more callback keys exist on the track timeline.

The next two lines of code might seem a bit confusing, so let us examine the time conversions which must be performed in this function in order to return the next callback key to the controller as a track position.

As we know from our earlier discussions, callback keys, just like SRT keys, are defined in ticks. Along with SRT keys, they help define the period of the animation set. If an animation set has a period of 10 seconds, then we know that any callback keys will have timestamps within this period. However, the controller is not interested in getting back a callback key timestamp as a periodic position because it has no concept of animation set local time. The controller wants to know when the next callback event will occur in track time, and as we know, the two are very different.

Let us imagine that we have a looping animation set with a period of 10 seconds which contains a single callback key at 10 seconds. We know that the track position will continue to increase with calls to AdvanceTime, but the periodic position of the animation set will loop every 10 seconds. If we simply searched for a callback key from the passed track position each time, once the track position exceeded 10 seconds (and continued to increase) we would never find another. The next time the animation set looped, our sound effect (for example) would not play. Therefore, we need to map the passed track position into a periodic position, perform the search for the next callback key starting from that periodic position, and, once found, convert that periodic position back into a future track position and return it to the controller.

With our 10 second animation set example, at a track position of 15 seconds, the callback key (at 10 seconds periodic position) would have been executed once. Now the controller would call the GetCallback method again to find another callback key, passing in a track position of 15 seconds. Our function would convert this into a periodic position of 5 seconds (i.e., 5 seconds into the second loop) and it would start the search and find our callback key at 10 seconds (periodic position).

Of course, we cannot simply return this periodic position of 10 seconds back to the controller because the track position is already at 15 seconds and it wants to know the next (future) time that this callback key needs to be executed in track time. So our function, having been passed a track position of 15 seconds, first maps this to a 5 second periodic position. We calculate that value in the prior code by performing the modulus of the passed track position and the period of the animation. This gives us:

PeriodicPosition = fmod (TrackPosition , Period) * TicksPerSecond
PeriodicPosition = fmod (15 , 10)
PeriodicPosition = 5

This calculation maps the track position to a periodic position. We know that whatever loop we are on, we are currently 5 seconds through it. You can see that in the above code listing, we perform this calculation and store the result in the local variable fPeriodic. This is the periodic position from which we wish to begin our search for callback keys. Notice that when we perform that calculation, we also multiply the result by m_fTicksPerSecond so that the periodic search start position is now specified in

ticks. Since all of our keys are specified with tick timestamps, we now have our start position using the same timing type as the callback keys for the search.

You might think that we now have everything we need, but this is not the case quite yet. While the periodic position certainly allows us to search and find the next scheduled callback key, its timestamp will be a periodic position. The controller will want to get that timestamp back as a track position, so the timestamp of the callback we eventually return must be converted from a periodic position into a track position. The problem is that normally the animation set would not have any idea which iteration of a loop it is on because it does not record how many times it has looped. However, this is precisely why the animation controller passes us a track position and not a periodic position (as it does with the GetSRT method). It is only when we have the track position from which to begin the search do we have the necessary information to turn the periodic timestamp of a key back into a track position.

If you look at the previous code listing you will see that we calculate a value and store it in a local variable called fLoopTime. This uses the passed track position and the previously calculated periodic position to get the number of complete loops the animation has performed.

In our example, we said the function was passed a track position of 15 seconds and the period of the animation set was 10 seconds. At this point we have also calculated the periodic position as being 5 seconds. Therefore, if we subtract the current periodic position from the track position, we have the number of seconds of complete animation loops that have been performed:

LoopTime = TrackPosition – PeriodicPosition
LoopTime = 15 – 5
LoopTime = 10 seconds of completed loops.

This is exactly what we do in the first section of the function code shown previously -- we simply subtract the periodic position from the track position. However, our periodic position is currently in ticks, so we convert the track position to ticks also. What ends up in the fLoopTime local variable is the number of ticks of complete loops that the animation has played out. This can also be thought of as the track position at which the current animation loop began (converted to ticks).

So we now know what we have stored in the two local variables. But why do we need know the total loop time? Well, in our example we said that the callback key was at a periodic position of 10 seconds. Therefore, we would begin our search from 5 seconds into the callback key array and find that the next callback key encountered has a periodic position of 10 seconds. We know that this is a periodic position, but the controller wants this to be a track position. That conversion is now very easy. We simply add the timestamp of the callback key to the total loop time (the track position at which the current loop began) and we have the track position in the future that this key is scheduled for:

FutureTrackPos = LoopTime + TimeStamp
= 10 + 10
= 20 seconds track time this key will be triggered

As you can see, by simply adding the timestamp of the key (which is a periodic position) to the total loop time, we end up with that future periodic position as a track time. The periodic position is

essentially like an offset from the track position when this current animation loop started. And we know that this works out properly. If we have a 10 second long looping animation set which has a single callback key at 10 seconds (periodic position), we know that the callback key should be triggered at 10 seconds, 20 seconds, 30 seconds, etc. (i.e., every time the track position crosses the 10 second boundary of the animation set's period).

That was quite a long discussion to explain two lines of code, but now we know what these local variables contain and how we need to map the resulting timestamp back into a track position. The rest of the function will be easy to understand.

The remainder of the function is mostly divided into two code blocks depending on whether the `D3DXCALLBACK_SEARCH_BEHIND_INITIAL_POSITION` flag has been specified by the controller. You will recall from the textbook that this flag instructs our function to perform the search for the next callback key from the initial position in a backwards direction instead of the more common forward direction.
We will take a look at the code that deals with finding the next callback key in forward search mode first since this is the mode used by the controller all of the time. In other words, this is the code that is executed if the `D3DXCALLBACK_SEARCH_BEHIND_INITIAL_POSITION` flag is not passed by the controller.

```
    // Searching forwards?
    if ( !(Flags & D3DXCALLBACK_SEARCH_BEHIND_INITIAL_POSITION) )
    {
        // Find the first callback equal to, or greater than the periodic position
        for ( i = 0; i < m_nCallbackCount ; ++i )
        {
            if ( m_pCallbacks[i].Time >= fPeriodic ) break;
        }// Next Callback

        // If nothing was found, then this HAS to be the first key
        // (for the next loop)
        if ( i == m_nCallbackCount )
        {
            // Increase the loop time and use the first key
            fLoopTime += (m_fLength * m_fTicksPerSecond);
            i = 0;

        } // End if wrap
```

This code block sets up a loop to iterate through each callback key in the callback key array. Inside the loop, we compare the timestamp of each callback key and break when we find the first callback key that has a timestamp that is either greater than or equal to the search start time. What we are doing here is simply trying to find the first callback key that is not behind the search start time. As this is our forward search mode, we are not interested in any callback keys that are behind the track position passed in, as they will have already been executed.

Once we find a suitable callback key, we break from the for loop. At this point the loop variable 'i' will contain the index of the key in the array that we are potentially interested in. You can see in the above code that once outside the loop, if 'i' is equal to the number of callback keys, then no future callback keys exist. We know then that the search start time was greater than the timestamps of all callback keys in the array. When this is the case, we essentially need to loop around the animation. So we know that the first key in the array is now the key we want as we have executed all keys right up to the end of the

array and it is time to start again from the beginning. However, we cannot just simply set ‘i’ to zero and leave it at that, since we need to account for the fact that we have just performed another loop of animation that we did not account for when we calculated the fLoopTime variable at the start of the function. Since we have searched to the end of the current period and have looped back around to the beginning again, we have just performed an additional animation loop. Therefore, we add onto the fLoopTime variable (which stores the current number of animation loops in ticks) the period of the animation set (in ticks).

For example:

AnimationSet.Period =10 seconds
CallBackKey.Time = 5 seconds
StartSearchTrackPosition = 18 seconds
StartSearchPeriodicPosition = 8 seconds
LoopTime = 10 seconds

The above settings demonstrate how this function might be called for an animation set with a period of 10 seconds that has a single callback key registered at 5 seconds. The function is passed a start search time (as a track position) of 18 seconds. We showed earlier how a modulus of this value with the period of the animation set will convert the search start time into a periodic position of 8 seconds.

Now we know, just by looking at the data described above, that from a track position of 18 seconds, the next callback key should actually be at track position 25 seconds (five seconds into the third loop). So we perform the search for the next callback key from 8 to 10 seconds and we do not find one; this is because the only callback key has a timestamp less than 8 seconds. This means we need to loop. We know that the first callback key is the callback key we are interested in, and we also increment the loop time by the period of the animation set:

LoopTime + = 10
LoopTime = 20 seconds

Now that we have compensated for the additional loop, later in the function we will be able to calculate the track position of the callback key as:

TrackPos = LoopTime + CallBackKey.Time;
= 20 + 5
= 25 seconds.

And there we have it. The addition to the loop time variable in the above code lets us keep a record of the number of animation loops performed. It is used later in the function to convert the timestamp of the callback key back into a future track position.

At this point we now have the index of the callback key stored in loop variable ‘i’. The next section of code (the conclusion of the forward search) fetches this key from the array and then performs several tests for compliance with the input search flags.

```
        // Get the key
        Key = m_pCallbacks[ i ];

        // Do we ignore keys that share the same position?
        bool bExclude = Flags & D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION;

        if ( (bExclude && Key.Time == fPeriodic) )
        {
            // If we're going to wrap onto the next loop, increase our loop time
            if ( (i + 1) == m_nCallbackCount )
              fLoopTime += (m_fLength * m_fTicksPerSecond);

            // Get the next key (taking care to loop round where appropriate)
            Key = m_pCallbacks[ (i + 1) % m_nCallbackCount ];

        } // End if skip to the next key

    } // End if searching FORWARDS
```

In the above code, once the callback key data is fetched from the array we test to see if the caller (the animation controller) specified `D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION`. If so, then we are not interested in this key if it has a timestamp that is equal to the search position. As you can see, if the flag is set and the key timestamp is equal to the periodic position then we need to ignore this key and fetch the next one in the array -- element i+1. Notice the conditional that states if i+1 is equal to the number of callback keys, then the original key was the last one in the callback array and we need to loop around and fetch the first callback key. If this is the case, we record this loop in the fLoopTime variable so that we always have an accurate record of exactly how many ticks of animation looping has been performed by this animation set. Finally, we fetch the actual key we need. Notice the code we use to index into the correct array element. We use the modulus of i+1 and the number of callback keys in the array. If i+1 is equal to the number of callback keys, zero is returned from the operation and the first callback key in the array is used as required.

We have now covered the code to find the correct callback key when we are in forward searching mode. The next section of code shows essentially the reverse operation being performed. We will not take this code block line by line as it is just a mirror image of the code we have just discussed. Rather than searching forward from the current search position and looping back around to the beginning when necessary, it searches backwards from the passed search position looping back around to the end of the array when necessary.

```
// else we are in backwards search mode
    else
    {
        // Find the first callback equal to, or greater than the periodic position
        for ( i = (signed)m_nCallbackCount - 1; i >= 0; --i )
        {
            if ( m_pCallbacks[i].Time <= fPeriodic ) break;

        } // Next Callback

        // If nothing was found, then this HAS to be the last key
        //(for the next loop)
        if ( i == -1 )
```

```
        {
            // Decrease the loop time and use the last key
            fLoopTime -= (m_fLength * m_fTicksPerSecond);
            i = (signed)m_nCallbackCount - 1;

        } // End if wrap

        // Get the key
        Key = m_pCallbacks[ i ];

        // Do we ignore keys that share the same position?
        bool bExclude = Flags & D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION;
        if ( (bExclude && Key.Time == fPeriodic) )
        {
            // If we're going to wrap onto the previous loop,
            // decrease our loop time
            if ( (i - 1) < 0 ) fLoopTime -= (m_fLength * m_fTicksPerSecond);

            // Get the next key (taking care to loop round where appropriate)
            Key = m_pCallbacks[ (i - 1) % (signed)m_nCallbackCount ];

        } // End if skip to the next key

    } // End if searching forwards
```

At this point in the function we have located our callback key. The controller wants us to store the track position of this callback key in the pCallbackPosition output parameter it passed in, so we assign the sum of the loop time and the timestamp of the callback key. Since both the loop time and the callback key timestamp are in ticks, we must divide the result by the TicksPerSecond ratio of the animation set so that it is a track position specified in seconds. We also assign the context data pointer of the callback key (which in our code will actually be a void pointer to a CActionData object) to the ppCallbackData pointer passed in by the contoller. The last section of the code that performs these steps is shown below.

```
    // Return the time, mapped to the track position (converted back to seconds)
    if ( pCallbackPosition )
        *pCallbackPosition = (Key.Time + fLoopTime) / m_fTicksPerSecond;

    // Return the callback data
    if ( ppCallbackData )
        *ppCallbackData = Key.pCallbackData;

    // Success!!
    return D3D_OK;
}
```

So we see that there is nothing magical about the functions that are implemented by ID3DXKeyframedAnimationSet. While we took a black-box view of them in the textbook, now we have had a chance to see exactly how the animation set passes back information to the animation controller in response to a next scheduled callback key search.

This concludes our coverage of the code and methods in our CAnimationSet class. We still have a ways to go yet though with respect to understanding the system functionality. Next we will need to look at the code that generates SRT data using interpolation. The keyframe interpolation code is tucked away in the

CAnimationItem class, so we will look at this class next to wrap up coverage of our custom animation set.

Source Code Walkthrough - CAnimationItem

For every animation in the set, the keyframe data will be stored in a CAnimationItem object. If our animation set contains 10 animations, it will have an array of 10 CAnimationItem objects. Each item in this array will contain the keyframe data for a single frame in the hierarchy and will expose the methods to build SRT data for any periodic position and return it to the animation set. The set then hands the SRT information back the animation controller.

While this class has only a few methods, it does contain one of the most important in our system -- GetSRT. As we saw in our coverage of CAnimationSet::GetSRT, that function acted as a wrapper around a call to the GetSRT method of a CAnimationItem object at the requested animation index.

Let us first look at the class declaration (see CActor.h).

```
class CAnimationItem
{
public:
    //-----------------------------------------------------------------------
    // Constructors & Destructors for This Class.
    //-----------------------------------------------------------------------
     CAnimationItem( );
    ~CAnimationItem( );

    // Public methods for the class
    LPCTSTR GetName     ( ) const;
    HRESULT GetSRT      ( DOUBLE PeriodicPosition, D3DXVECTOR3 *pScale,
                          D3DXQUATERNION *pRotation, D3DXVECTOR3 *pTranslation );
    HRESULT BuildItem   ( ID3DXKeyframedAnimationSet *pAnimSet, ULONG ItemIndex );

private:
    LPTSTR              m_strName;              // The name of this animation
    D3DXKEY_VECTOR3    *m_pScaleKeys;           // The scale keys
    D3DXKEY_VECTOR3    *m_pTranslateKeys;       // The translation keys
    D3DXKEY_QUATERNION *m_pRotateKeys;          // The rotation keys
    ULONG               m_nScaleKeyCount;       // Number of scale keys
    ULONG               m_nTranslateKeyCount;   // Number of translation keys
    ULONG               m_nRotateKeyCount;      // Number of rotation keys
    double              m_fTicksPerSecond;      // Source ticks per second

    ULONG               m_nLastScaleIndex;     // Cache for last 'start' scale key
    ULONG               m_nLastRotateIndex;    // Cache for last 'start' rot key
    ULONG               m_nLastTranslateIndex; // Cache for last 'start' trans key
    double              m_fLastPosRequest;     // Cache for last periodic position
};
```

The CAnimationItem class exposes three methods. The GetName function returns the name of the animation, which will also match the name of the frame in the hierarchy that it animates. The GetSRT method is responsible for generating the SRT data for a given periodic position. The BuildItem method is used by our CAnimationSet class constructor to copy over the keyframe data from the original animation in the ID3DXKeyframedAnimatonSet.

You will recall how the CAnimationSet constructor is passed a pointer to an ID3DXKeyframedAnimationSet which it will essentially clone. The code fetches the number of animations contained in the animation set and then uses this value to allocate a CAnimationItem array of the same size. The code then looped through each CAnimationItem and called its CAnimationItem::BuildItem function. The function is passed the interface of the ID3DXKeyframedAnimationSet and the index of the animation which we would like to copy into this CAnimationItem. As a reminder, here is a small excerpt from the CAnimationSet constructor which we discussed earlier:

```
// Allocate memory for animation items
    m_pAnimations = NULL;
    m_nAnimCount  = pKeySet->GetNumAnimations();
    if ( m_nAnimCount )
    {
        // Allocate memory to hold animation items
        m_pAnimations = new CAnimationItem[ m_nAnimCount ];
        if ( !m_pAnimations ) throw E_OUTOFMEMORY;

        // Build the animation items
        for ( i = 0; i < m_nAnimCount; ++i )
        {
            hRet = m_pAnimations[ i ].BuildItem( pKeySet, i );
            if ( FAILED(hRet) ) throw hRet;

        } // Next animation

    } // End if any animations
```

The BuildItem function is responsible for extracting the keyframe data from the specified animation in the ID3DXKeyframedAnimationSet and copying it into internal SRT arrays. We will see this function in a moment.

Let us now discuss the members of the CAnimationItem class.

LPTSTR m_strName

This member points to a string that contains the name of the animation. There will also be a frame in the hierarchy with a matching name. The name of the frame and the animation essentially form a link for the controller between the animation's keyframe and the frame matrix in the hierarchy it is to animate.

D3DXKEY_VECTOR3 *m_pScaleKeys

If this animation contains any scale keyframes, then this will point to an array of D3DXKEY_VECTOR3 keys. The array will be allocated in the BuildItem method when the scale key

list is copied over from the matching animation in the ID3DXKeyframedAnimationSet. If no scale keyframes are defined for this animation, this member will be set to NULL.

D3DXKEY_VECTOR3 *m_pTranslateKeys
If this animation contains any positional keyframes, then this will point to an array of D3DXKEY_VECTOR3 keys. The array will be allocated in the BuildItem method when the translation key list is copied over from the matching animation in the ID3DXKeyframedAnimationSet. If no translation keyframes are defined for this animation, this member will be set to NULL.

D3DXKEY_QUATERNION *m_pRotateKeys
If this animation contains any rotation keyframes, then this will point to an array of D3DXKEY_QUATERNION keys. The array will be allocated in the BuildItem method when the rotational keyframe list is copied over from the matching animation in the ID3DXKeyframedAnimationSet. If no scale keyframes are defined for this animation, this member will be set to NULL.

The three arrays just described contain the SRT keyframes of the animation. These are the keyframes that will be used for the interpolation and generation of SRT data in the GetSRT function.

ULONG m_nScaleKeyCount
ULONG m_nTranslateKeyCount
ULONG m_nRotateKeyCount
These three members will be set in the BuildItem method. They describe the number of scale keys, translation keys, and rotation keys stored in the m_pScaleKeys, m_pTranslateKeys and m_pRotateKeys arrays respectively.

double m_fTicksPerSecond
This member will store the TicksPerSecond ratio of its parent animation set. We will need to access this value frequently during interpolation in the GetSRT method, so we copy this value from the parent animation set and cache it for ease of access. Remember that this is a per-set property not a per animation property, so all CAnimationItems in the same CAnimationSet will have the same value.

ULONG m_nLastScaleIndex
ULONG m_nLastRotateIndex
ULONG m_nLastTranslateIndex
These three members are used to optimize the GetSRT method. They are stored so that we do not have to loop through all keyframes in all three arrays to find the bounding keyframes for the passed periodic position every time GetSRT is called by the controller. For example, for periodic position A, we might find that the two bounding scale keyframes are at indices m_pScaleKeys[25] and m_pScaleKeys[26]. What we will do at this point is store the index 25 in the m_nLastScaleIndex member. The next time GetSRT is called for this animation for periodic position B, we can assume a forward moving timeline and begin the search for bounding keyframes starting at scale key 25. This optimizes our search for keyframes by ignoring those that are behind the last cached position and reduces loop traversal overhead.

It should be noted that this is an efficient optimization only when the timeline is being advanced in a forward direction (which is almost all of the time). It provides no speed benefit when playing the animation in the reverse direction (e.g., in ping pong mode where every other loop the periodic position is stepping backwards). We must also make sure that we invalidate the cache as soon as the current periodic position wraps around and becomes smaller than the last cached key time. This is all handled in the GetSRT method, which we will look at in a moment.

double m_fLastPosRequest

This final member is used to store the periodic position of a previous call to the GetSRT method. For example, when the GetSRT method is called, the current periodic position is stored in this member. The next time the GetSRT method is called, we can test the new periodic position against the previous one. If the new periodic position is less than the previous position, we know that the animation has looped (or moved backwards) and we can set the three cached indices (m_nLastScaleKey, m_nLastRotateIndex and m_nLastTranslateIndex) to zero. This make sure that when looping occurs, the search for bounding keyframes always begins at keyframe 0 in each array the next time through.

Let us now examine the three methods exposed by this object. Ww will begin with the CAnimationItem::BuildItem method.

CAnimationItem::BuildItem

BuildItem is called by the CAnimationSet constructor for each CAnimationItem created. The job of this function is to populate the CAnimationItem's internal arrays with the SRT data from the ID3DXKeyframedAnimationSet that is in the process of being cloned and replaced. Since a CAnimationItem object represents the data for a single animation in the set, it is passed the interface to the keyframed animation set that needs to be copied and the index of the animation within that set that will be copied.

The function first tests that the index passed in is within the number of animations contained in the source keyframed animation set. It exits if this is not the case. If the animation index is valid, we use the ID3DXKeyframedAnimationSet::GetAnimationNameByIndex method to fetch the name of the animation within the animation set. As you can see in the following code, we call this method passing in the index of the animation within the set that we wish to retrieve the name for, and a pointer to a string that will point to the name of the animation on function return. We use the tcsdup method to duplicate the string so that we can store a copy of the animation name in the CAnimationItem::m_strName member variable. This name should match the name of a frame in the hierarchy and the name of a registered animation output.

```
HRESULT CAnimationItem::BuildItem( ID3DXKeyframedAnimationSet *pAnimSet,
                                   ULONG ItemIndex )
{
    LPCTSTR strName = NULL;

    // Validate
    if ( !pAnimSet || ItemIndex > pAnimSet->GetNumAnimations( ) )
        return D3DERR_INVALIDCALL;
```

```
    // Get the name and duplicate it
    pAnimSet->GetAnimationNameByIndex( ItemIndex, &strName );
    m_strName = _tcsdup( strName );
```

In the next section we retrieve the number of scale, rotation and translation keys that exist for the animation we are currently in the process of copying. We will obviously need these values to allocate the SRT arrays of the CAnimationItem and to loop through the keyframes in the GetSRT method. We also extract the TicksPerSecond ratio from the animation set and store that in a CAnimationItem member variable. This is an animation set value that will be consistent for all animations in the set, and we store a copy of it inside each CAnimationItem so that we can easily access it from GetSRT without having to query into the parent animation set.

```
    // Store any secondary values
    m_fTicksPerSecond    = pAnimSet->GetSourceTicksPerSecond( );
    m_nScaleKeyCount     = pAnimSet->GetNumScaleKeys( ItemIndex );
    m_nRotateKeyCount    = pAnimSet->GetNumRotationKeys( ItemIndex );
    m_nTranslateKeyCount = pAnimSet->GetNumTranslationKeys( ItemIndex );
```

At this point, we know how many keyframes exist for this animation in each of its scale, rotation, and translation arrays. We will need to allocate these arrays in our CAnimationItem if we intend to copy and store this data from the ID3DXKeyframedAnimationSet. In the following code you can see that we use the values just retrieved to allocate the SRT arrays:

```
    // Allocate any memory required
    if ( m_nScaleKeyCount )
        m_pScaleKeys  = new D3DXKEY_VECTOR3[ m_nScaleKeyCount ];

    if ( m_nRotateKeyCount)
        m_pRotateKeys = new D3DXKEY_QUATERNION[ m_nRotateKeyCount ];

    if ( m_nTranslateKeyCount )
        m_pTranslateKeys = new D3DXKEY_VECTOR3[ m_nTranslateKeyCount ];
```

With our arrays now allocated, all that is left to do is copy the keyframe data from the animation in the ID3DXKeyframedAnimationSet into our own SRT arrays. Luckily, ID3DXKeyframedAnimationSet provides an easy means for retrieving this data. We can retrieve the scale, rotation or translation keys simply by calling the GetScaleKeys, GetRotationKeys, and GetTranslationKeys methods. We pass these functions the index of the animation that we wish to retrieve the keyframes for (the first parameter), and a pointer to a pre-allocated array that will receive the copy of the keyframe data (the second parameter). These arrays are the ones we just allocated. The functions will then copy the keyframe data from the ID3DXKeyframedAnimationSet directly into our CAnimationItem SRT arrays.

```
    // Retrieve the keys
    if ( m_pScaleKeys     )
        pAnimSet->GetScaleKeys( ItemIndex, m_pScaleKeys );

    if ( m_pRotateKeys    )
        pAnimSet->GetRotationKeys( ItemIndex, m_pRotateKeys );
```

```
    if ( m_pTranslateKeys )
        pAnimSet->GetTranslationKeys( ItemIndex, m_pTranslateKeys );

    // We're done!
    return D3D_OK;
}
```

At this stage, our CAnimationItem contains all SRT keyframe data for a single frame in the hierarchy. When the CAnimationSet constructor has called this method for each animation, we will have copied all the animation data from the original keyframed animation set. We can then use our CAnimationSet instead of the original ID3DXKeyframedAnimationSet and sidestep the bug in the ID3DXKeyframedAnimationSet::GetSRT method.

CAnimationItem::GetName

The second method of the CAnimationItem class is a utility function which returns the name of the animation item. It is vital that this function be available to our CAnimationSet class. Recall that our CAnimationSet class must provide a function called GetAnimationNameByIndex (part of the ID3DXAnimationSet base interface). The animation controller will expect this function to be present in our animation set so that it can be called to fetch the name of an animation at the specified index. In our implementation, animations are stored in a CAnimationItem array inside the CAnimationSet. The CAnimationSet::GetAnimationNameByIndex can return the name of an animation to the animation controller by using the passed array index to call CAnimationItem::GetName.

```
LPCTSTR CAnimationItem::GetName( ) const
{
    return m_strName;
}
```

CAnimationItem::GetSRT

This method is, in many ways, the backbone of the entire D3DX animation system. It is responsible for generating the SRT animation data for a specified periodic position.

The animation controller will call the GetSRT method of an animation set for each animation in that set. Each time, the animation index will be passed and the function will generate the SRT data for that animation. This information will be returned to the controller and used to rebuild the frame matrix which shares the same name as the animation.

Although the animation controller expects the call to the GetSRT method of the animation set interface to generate the SRT data, we saw earlier that the animation set GetSRT method acts as a wrapper that forwards the request to the correct CAnimationItem:

```
    return m_pAnimations[ AnimationIndex ].GetSRT( PeriodicPosition, pScale,
                                                   pRotation, pTranslation );
```

As the above code snippet demonstrates, the CAnimationSet::GetSRT method uses the controller supplied index (identifying which animation it needs SRT data for) and periodic position to call the GetSRT method of the corresponding CAnimationItem. CAnimationItem::GetSRT performs the keyframe interpolation for the specified time and stores the final scale, rotation, and translation information in the output parameters.

We took a look at how to implement a GetSRT function in the textbook, so the code to this function should look very familiar to you. Since the function is rather large, we will discuss it a section at a time.

The first section is shown below followed by a discussion of the code.

```
HRESULT CAnimationItem::GetSRT( DOUBLE PeriodicPosition, D3DXVECTOR3 *pScale,
                                D3DXQUATERNION *pRotation,
                                D3DXVECTOR3 *pTranslation )
{
    ULONG                i;
    D3DXQUATERNION       q1, q2;
    double               fInterpVal, fTicks;
    LPD3DXKEY_VECTOR3    pKeyVec1 , pKeyVec2;
    LPD3DXKEY_QUATERNION pKeyQuat1, pKeyQuat2;

    // Validate parameters
    if ( !pScale || !pRotation || !pTranslation ) return D3DERR_INVALIDCALL;

    // Clear them out as D3D does for us :)
    *pScale       = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    *pTranslation = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    D3DXQuaternionIdentity( pRotation );

    // Reset index caching if we've wrapped around
    if ( PeriodicPosition < m_fLastPosRequest )
    {
        m_nLastScaleIndex       = 0;
        m_nLastRotateIndex      = 0;
        m_nLastTranslateIndex   = 0;

    } // End if wrapped / move backwards

    // Store the last requested position
    m_fLastPosRequest = PeriodicPosition;

    // Now calculate the 'TimeStamp' value.
    fTicks = PeriodicPosition * m_fTicksPerSecond;
```

The function is called by CAnimationSet::GetSRT and is passed the periodic position of the parent animation set along with the addresses of two 3D vectors and a quaternion which the function will populate with the interpolated SRT data. If any of these pointers are NULL, then the function returns immediately, reporting an invalid call.

Next, we set the passed scale and translation vectors to zero and set the rotation quaternions to identity. This is done for safety -- the animation might not have S, R, or T arrays defined, so default values will need to be returned.

The next portion of the above code then tests to see if the current periodic position is smaller than the periodic position that was stored in the previous call. If so, then the periodic position has wrapped around (looped or ping-ponged). It is very important for us to trap this occurance because it affects our bounding keyframe search optimization. Recall that we store the first of the two bounding keyframes (from each array) so that the next time the function is called, we do not have to start looping through the keyframes from the first element. For example, if the first time this function was called, we found the two bounding scale keyframes to be at positions 20 and 21 in the scale keyframes array, the value of 20 (the low bounding keyframe) would be stored in the m_nLastScaleIndex member variable. This means that the next time GetSRT is called for this animation, we can begin searching for bounding keyframes from position 20 in the array instead of 0, thereby skipping tests for the first 20 keyframes in the scale keyframe array.

As you can see, we store a ‘last index’ for each of the arrays (scale, rotation and translation). This optimization assumes a forward moving timeline, so it expects the next call to GetSRT to have a greater periodic position than the previous one. However, when the periodic position of the animation set loops around to zero again, this will not be the case.

Imagine that the first time GetSRT is called the periodic position is such that the low bounding scale keyframe used is at index 99 in our scale keyframe array. At the end of this function, we would store this index in the m_nLastScaleIndex member. Now imagine that the periodic position loops around the next time this function is called such that the low bounding keyframe should now be at index 0. If we simply started searching from index 20 to the end of the array we would never find any bounding scale keyframes. Therefore, you can see that we also cache the previous periodic position. If it is larger than the current periodic position passed into the function, it means that the periodic position has either looped around to zero or, in the case of ping-pong mode, the periodic position is moving backwards. When this is the case, we must start searching for bounding keyframes in each of our arrays from position zero again to make sure we test them all. We do this by setting the three previous keyframe index caches to zero.

The implication is that every time the periodic position loops around, we will not benefit from our ‘last cached index’ optimization. It also means that this optimization will provide no benefit for animation sets that are moving backwards. If the current periodic position has not looped around such that it is smaller than the previous position, no action is taken. You will see in a moment how the cached last keyframe indices will be used to efficiently search through the keyframe arrays.

Next we set the current periodic position in the m_fLastPosRequest member so that the next time this function is called, we will be able to detect a periodic position wrap-around (as just discussed).

The final section of the previous code takes the input periodic position and multiplies it by the TicksPerSecond ratio so that we have the periodic position in ticks rather than seconds. This step is important because the keyframe timestamps are defined in ticks and we want to find the two keyframes in each array that bound the periodic position in ticks, not seconds. Because we took the liberty of

storing the TicksPerSecond ratio of the parent animation set in a CAnimatonItem member variable, we can access it easily without having to implement a CAnimationSet accessor function to return this value every time we need it.

The next section of code is responsible for searching for the bounding keyframes in the animation's scale keyframe array. Notice that we do not simply loop from the start of the keyframe array every time, but start at the m_nLastScaleIndex element in the array. As discussed, this index will usually store the low bounding scale keyframe index from a previous call to the function. If the periodic position has looped around however, this value will be set to zero and the scale keyframe array is searched from the beginning. Notice that we also use two D3DXKEY_VECTOR3 pointers to point to the two bounding keyframes that we find.

```
    // ******************************************************************
    // * SCALE KEYS
    // ******************************************************************
    pKeyVec1 = pKeyVec2 = NULL;
    for ( i = m_nLastScaleIndex; i < m_nScaleKeyCount - 1; ++i )
    {
        LPD3DXKEY_VECTOR3 pKey     = &m_pScaleKeys[i];
        LPD3DXKEY_VECTOR3 pNextKey = &m_pScaleKeys[i + 1];

        // Do these keys  bound the requested time ?
        if ( fTicks >= pKey->Time && fTicks <= pNextKey->Time )
        {
            // Update last index
            m_nLastScaleIndex = i;

            // Store the two bounding keys
            pKeyVec1 = pKey;
            pKeyVec2 = pNextKey;
            break;

        } // End if found keys

    } // Next Scale Key
```

In the above code we loop and fetch the current scale keyframe and the one that follows it in the array. We then compare the periodic position in ticks (fTicks) to see if it is greater than or equal to the current keyframe's timestamp and less than or equal to the next keyframe's timestamp. If this condition is true, then we have found the two keyframes in the array that bound the current periodic position. When this is the case, we store the index of the current keyframe 'i' in the m_nLastScaleIndex member so that the next time this function is called, we can begin searching from this index. We then assign the local variables pKeyVec1 and pKeyVec2 to point to these two scale keyframes. Our search is complete and we have found the two bounding scale keyframes, so we exit the loop. The next step will be to interpolate between these two keyframes to generate a single scale vector that can be returned back to the controller and contribute to the building of the associated frame matrix.

As discussed in the textbook, we can use the current periodic position (in ticks) to perform a linear interpolation (or 'lerp' for short) between the two vectors. First we subtract the periodic position from the timestamp of the low bounding keyframe. Then we divide this value by the difference between the

two timestamps of the keyframes. This will result in a value between 0.0 and 1.0 which describes, as a percentage, the periodic position between these two keyframes. If a value of 0.25 was generated, it would mean that actual periodic position is ¼ of the way between timestamp one and timestamp two.

We feed this value into the D3DXVec3Lerp function which will return a vector that is a weighted interpolation of the two vectors. Using our 0.25 example, D3DXVec3Lerp would return a scale vector that is comprised of ¾ of keyframe one and ¼ of keyframe two. The basic formula for a lerp is ***Vector1 + t(Vector2 – Vector2)*** where **t** is our interpolation value between zero and one. The resulting interpolated scale vector is stored in the pScale vector that was passed in as a parameter by the animation controller. Below we see the code that performs this task:

```
    // Make sure we found keys
    if ( pKeyVec1 && pKeyVec2 )
    {
        // Calculate interpolation
        fInterpVal  = fTicks - pKeyVec1->Time;
        fInterpVal /= (pKeyVec2->Time - pKeyVec1->Time);

        // Interpolate!
        D3DXVec3Lerp( pScale,
                      &pKeyVec1->Value,
                      &pKeyVec2->Value,
                      (float)fInterpVal );
    } // End if keys were found
    else
    {
        // Scale is the same as the last scale key found
        if ( m_nScaleKeyCount )
          *pScale = m_pScaleKeys[ m_nScaleKeyCount - 1 ].Value;

        // Inform cache that it should no longer
        // search unless cache is invalidated
        m_nLastScaleIndex = m_nScaleKeyCount - 1;


    } // End if no keys found
```

If pKeyVec1 or pKeyVec2 are NULL then the two bounding keyframes could not be found in the scale array. For example, this might happen if the periodic position is outside the range of defined scale keys. When this is the case the ‘else’ block of the above code is executed. In this code block we first test if any scale keys even exist in this object and if so, then we assume that the periodic position is larger than the last defined scale keyframe and we just copy the last scale vector defined in the SRT array into pScale. Essentially this amounts to freezing the scaling animation at the last keyframe. Any periodic positions past this timestamp will generate the same final scale vector. We then set the last scale keyframe index to the final keyframe in the scale array so that the next time the function is called, we will not waste time looping and testing any scale keys -- we will just return the final scale key. We are basically stating that this animation appears to be over so we will just return the last scale keyframe unless the periodic position wraps around. If the periodic position does loop around, then we know that the last index caches are cleared and the entire keyframe array is used for bounding keyframe searches again.

In the next section of the code, we perform nearly an identical search and interpolation to generate the rotation quaternion. We search the rotation keyframe array for the two bounding keyframes and once found, perform a spherical linear interpolation (or 'slerp' for short) using the periodic position. This generates a final quaternion that is stored in the pRotation output parameter passed into the function by the animation controller. Here is the code that searches the quaternion array for the two bounding keyframes:

```
    // ******************************************************************
    // * ROTATION KEYS
    // ******************************************************************
    pKeyQuat1 = pKeyQuat2 = NULL;
    for ( i = m_nLastRotateIndex; i < m_nRotateKeyCount- 1; ++i )
    {
        LPD3DXKEY_QUATERNION pKey     = &m_pRotateKeys[i];
        LPD3DXKEY_QUATERNION pNextKey = &m_pRotateKeys[i + 1];

        // Do these keys bound the requested time ?
        if ( fTicks >= pKey->Time && fTicks <= pNextKey->Time )
        {
            // Update last index
            m_nLastRotateIndex = i;

            // Store the two bounding keys
            pKeyQuat1 = pKey;
            pKeyQuat2 = pNextKey;
            break;

        } // End if found keys

    } // Next Rotation Key
```

Now that we have our two keyframes we perform a slerp between them. But first we flip the axes of our quaternion keyframes so that they are left-handed to match our coordinate system. The D3DXQuaternionConjugate function is used for that purpose. It takes the quaternion (x,y,z,w) and returns the quaternion conjugate (-x,-y,-z, w). We can now generate our fInterpVal value by mapping the periodic position with respect to the two keyframe timestamps into the 0.0 to 1.0 range and feed this value into the slerp function to generate the final rotation quaternion that the animation controller needs.

```
    // Make sure we found keys
    if ( pKeyQuat1 && pKeyQuat2 )
    {
        // Reverse 'winding' of these values
        D3DXQuaternionConjugate( &q1, &pKeyQuat1->Value );
        D3DXQuaternionConjugate( &q2, &pKeyQuat2->Value );

        // Calculate interpolation
        fInterpVal  = fTicks - pKeyQuat1->Time;
        fInterpVal /= (pKeyQuat2->Time - pKeyQuat1->Time);

        // Interpolate!
        D3DXQuaternionSlerp( pRotation, &q1, &q2, (float)fInterpVal );

    } // End if keys were found
```

```
    else
    {
        // Rotation is the same as the last key found
        if ( m_nRotateKeyCount )
        D3DXQuaternionConjugate(pRotation,
                                &m_pRotateKeys[ m_nRotateKeyCount - 1 ].Value );

        // Inform cache it should no longer search unless cache is invalidated
        m_nLastRotateIndex = m_nRotateKeyCount - 1;

    } // End if no keys found
```

As was the case with the scale array, we also have an ‘else’ code block that is executed if no bounding rotation keyframes are found. When this is the case the same optimization logic is applied. We simply return the conjugate of the last rotation keyframe in the array and set the m_nLastRotateIndex to point at the last element in the array. Subsequent calls to the function will no longer need to search for bounding quaternion keyframes and can simply return the final quaternion in the array. Only when the periodic position is reversed or looped will the last index cache be invalidated and searches through the quaternion array re-enabled.

The final section of the code searches for the two bounding translation keyframes and returns the interpolated position vector. This code is almost identical to the code that finds and interpolates between scale keyframes (they are both D3DXKEY_VECTOR3 arrays) so we should now fully understand the following code without discussion.

```
    // ******************************************************************
    // * TRANSLATION KEYS
    // ******************************************************************
    pKeyVec1 = pKeyVec2 = NULL;
    for ( i = m_nLastTranslateIndex; i < m_nTranslateKeyCount - 1; ++i )
    {
        LPD3DXKEY_VECTOR3 pKey     = &m_pTranslateKeys[i];
        LPD3DXKEY_VECTOR3 pNextKey = &m_pTranslateKeys[i + 1];

        // Do these keys bound the requested time ?
        if ( fTicks >= pKey->Time && fTicks <= pNextKey->Time )
        {
            // Update last index
            m_nLastTranslateIndex = i;

            // Store the two bounding keys
            pKeyVec1 = pKey;
            pKeyVec2 = pNextKey;
            break;

        } // End if found keys

    } // Next Translation Key

    // Make sure we found keys
    if ( pKeyVec1 && pKeyVec2 )
    {
        // Calculate interpolation
```

```
        fInterpVal  = fTicks - pKeyVec1->Time;
        fInterpVal /= (pKeyVec2->Time - pKeyVec1->Time);

        // Interpolate!
        D3DXVec3Lerp( pTranslation,
                      &pKeyVec1->Value,
                      &pKeyVec2->Value,
                      (float)fInterpVal );

    } // End if keys were found
    else
    {
        // Rotation is the same as the last key found
        if ( m_nTranslateKeyCount )
             *pTranslation = m_pTranslateKeys[ m_nTranslateKeyCount - 1 ].Value;

        // Inform cache it should no longer search unless cache is invalidated
        m_nLastTranslateIndex = m_nTranslateKeyCount - 1;

    } // End if no keys found

    // We're done
    return D3D_OK;
}
```

We have now implemented all the components of our custom animation set. We examined how to build the CAnimationSet class which will be registered with the animation controller and we also discussed the CAnimationItem class that will be used by CAnimationSet to store, manage and interpolate keyframe data. CAnimationSet will be used in many of our demos from this point onwards, so be sure that you understand how it all works. Although the implementation of our ID3DXAnimationSet derived object may not have been something you thought we would have to do, the ID3DXKeyframedAnimationSet bug left us little choice. However, this has proven to be a beneficial aside as it has allowed us to get a much deeper insight into how the D3DX animation system works under the hood. It also provides you with some foundation material should you choose to implement other forms of custom animations that use application generated data.

Now that we have our drop-in replacement for the ID3DXKeyframedAnimationSet, let us return to our discussions concerning adding animation support to our CActor class. These features will be used in Lab Project 10.1 to play back an animated X file.

Source Code Walkthrough - CActor

Adding animation support to CActor will be a trivial task for the most part. While many functions will be added to the class in this lab project, virtually all of them will be simple wrappers around calls to the animation controller. Since these methods wrap the animation controller methods discussed in detail in the textbook, they will not require extensive coverage. Most are functions that set or retrieve properties from the actor's animation controller. This allows an application using our CActor class to access all the methods of its underlying animation controller via the CActor interface.

Below we see an excerpt from CActor.h which shows the new member variables that have been added in this lab project. For the sake of readability, we have not listed method declarations because we will cover the methods later. For now, just observe the new member variables which have been highlighted in bold.

```
class CActor
{
public:
        //---------------- Member Functions Go Here( Not Shown ) ---------------
private:
    LPD3DXFRAME                 m_pFrameRoot;               // Root 'frame'
    LPD3DXANIMATIONCONTROLLER   m_pAnimController;          // controller
    LPDIRECT3DDEVICE9           m_pD3DDevice;               // Direct3D Device .
    CALLBACK_FUNC               m_CallBack[CALLBACK_COUNT]; // callbacks
    TCHAR                       m_strActorName[MAX_PATH];   // Actor Filename

    D3DXMATRIX                  m_mtxWorld;      // Currently active world matrix.
    ULONG                       m_nOptions;      // The requested mesh options.

    // Limits of the Animation Mixer
    ULONG                       m_nMaxTracks;
    ULONG                       m_nMaxAnimSets;
    ULONG                       m_nMaxAnimOutputs;
    ULONG                       m_nMaxEvents;
    ULONG                       m_nMaxCallbackKeys;
};
```

Not surprisingly, we have now added an ID3DXAnimationController pointer to our CActor. It will be assigned to the animation controller returned from the D3DXLoadMeshHierachyFromX function (see LoadActorFromX).

Four of the five new members at the bottom of the listing contain the limits of the animation controller currently being used by the actor. Note that m_nMaxCallbackKeys is not a limit of the animation controller; it is used by the application to tell the actor the maximum number of callback keys that it will register with a single animation set. Earlier, we talked about how this value will be used when the actor is loading an X file. It indicates the size of the temporary array of callback keys that will be passed to the callback key callback function. This is the array that the application will fill with callback key data and return to the actor. We will see this all happening in a moment.

We know from earlier discussion that the animation controller limits must be defined at creation time. Such limits define the maximum number of tracks that can be used on the animation mixer, the maximum number of animation sets that can be simultaneously registered with the controller, the number of matrices/frames that can be manipulated by the controller, and the maximum number of sequencer events that can be registered on the global timeline. In the textbook we also discussed that when we use D3DXLoadMeshHierarchyFromX to load an animated X file, D3DX will create the animation controller for us. This means that setting these controller properties at creation time is out of our hands in that case. The maximum limits will all be specified automatically when the X file is loaded.

When D3DXLoadMeshHierarchyFromX is used to load an X file, an animation controller will be returned and stored by the actor if animation data was contained in the file. The maximum number of

tracks on the controller will always be set to two. Having only two tracks to use for blending (i.e., only two animation sets can be simultaneously played and blended at any one time) might prove to be too limiting in certain cases. You will see in a moment that the CActor will expose a method to allow us to ratchet up these limits either before or after the X file has been loaded and the animation controller has been created. Behind the scenes, the CActor will clone the old animation controller (if one previously exists) to create a new one with the desired maximum limits, while maintaining the integrity of the original controller's animation data. Ultimately this means that the application is not restricted to the default limits of the animation controller created by D3DXLoadMeshHierarchyFromX.

While there are many new functions which are simple one-line wrappers around animation controller interface calls, a number of significant changes have taken place in functions that previously existed (e.g., LoadActorFromX and some of its new helper functions). These methods have been upgraded from the previous lab project to create, store, and potentially clone the animation controller if the limits of the controller are not as desired. Thus, we will cover the code changes to these methods as well.

An intuitive place to start examining the changes to CActor is in its constructor.

CActor::CActor()

By default, we set the animation controller pointer to NULL as this will not receive a meaningful assignment until the X file is loaded. This will happen when the application calls the CActor::LoadActorFromX method.

We also set the default maximum limits for the animation controller. For consistency, we set them to the same defaults used by the D3DXLoadMeshHierarchyFromX function. You will see in a moment how these variables can be changed by the application.

By default, we set the maximum number of animation sets the controller will support to 1, although this will be assigned a proper value as soon as the X file is loaded.

```
CActor::CActor()
{
    // Reset / Clear all required values
    m_pFrameRoot      = NULL;
    m_pAnimController = NULL;
    m_pD3DDevice      = NULL;
    m_nOptions        = 0;

    // Setup the defaults for the maximum controller properties
    // (Defaults to the same defaults D3DX provides).
    m_nMaxAnimSets      = 1;
    m_nMaxAnimOutputs   = 1;
    m_nMaxTracks        = 2;
    m_nMaxEvents        = 30;
    m_nMaxCallbackKeys  = 1024;

    ZeroMemory( m_strActorName, MAX_PATH * sizeof(TCHAR) );
    D3DXMatrixIdentity( &m_mtxWorld );

    // Clear structures
```

```
    for ( ULONG i = 0; i < CALLBACK_COUNT; ++i )
        ZeroMemory( &m_CallBack[i], sizeof(CALLBACK_FUNC) );
}
```

Notice that we also set the maximum number of callback keys that can be registered with an animation set to 1024. This will probably be enough for most cases, but if this is not the case, we will be able to change this setting using the SetActorLimits method which we will discuss shortly. Just as we did in the previous lab project, we also set the world matrix of the actor to an identity matrix and zero out the name of the actor.

Finally, we loop through the actor's callback function array and set all elements to zero. This array previously contained only the callback functions for texture and materials, but we now have four elements in this array, with the fourth element reserved for registration of the callback key collection callback function. The actor defines the following enumeration for callback functions. It tells us the number of possible callbacks as well as the array indices for the callback function pointers.

```
enum CALLBACK_TYPE { CALLBACK_TEXTURE = 0,
                     CALLBACK_EFFECT = 1,
                     CALLBACK_ATTRIBUTEID = 2,
                     CALLBACK_CALLBACKKEYS = 3,
                     CALLBACK_COUNT = 4 };
```

The fourth array element stores a function pointer (and context data pointer) to an application defined callback function that will fill the callback key array for a given animation set. This callback function was discussed earlier in this workbook. As before, these callbacks are registered using the actor's RegisterCallback function which is unchanged from the previous lab project.

CActor::Release()

As in our previous lab project, the actor's Release method is called by the destructor to release all memory allocated by the actor. It may also be called by the application or from other actor methods when the data currently managed by the actor needs to be cleaned out or reset. It releases the frame hierarchy, the animation controller, and the device interface that it has stored. The two new function calls that have been added to this function are shown below in bold.

```
void CActor::Release()
{
    CAllocateHierarchy Allocator( this );

    // Release any active callback data
    ReleaseCallbackData( );

    // Release objects (notice the specific method for releasing the root frame)
    if ( m_pFrameRoot      ) D3DXFrameDestroy( m_pFrameRoot, &Allocator );
    if ( m_pAnimController ) m_pAnimController->Release();
    if ( m_pD3DDevice      ) m_pD3DDevice->Release();

    // Reset / Clear all required values
    m_pFrameRoot      = NULL;
```

```
    m_pAnimController = NULL;
    m_pD3DDevice      = NULL;
    m_nOptions        = 0;

    ZeroMemory( m_strActorName, MAX_PATH * sizeof(TCHAR) );

    // Since 'Release' is called just prior to loading, we should not
    // clear ANYTHING which is designed to be setup and stored prior
    // to the user making that load call. This includes things like
    // the world matrix, and the callbacks.
}
```

Releasing the animation controller interface is of course a logical thing to do when the actor is destroyed. Its release will, in turn, destroy all of its animation sets. However, before this is done, we must make sure that the callback data we have assigned to our animation sets is released. Remember, each callback key in our lab project will have a CActionData object assigned to its context data pointer. When the animation controller releases the animation sets it manages, it will simply release the callback array contained inside that animation set. Since this is the only place the CActionData pointers are stored, we will lose any ability after that point to de-allocate the memory containing the CActionData objects that were allocated in the callback key callback function at load time. Therefore, before we release the controller, the Release method makes a call to a new member function called ReleaseCallbackData. This function (covered next) will release the memory for each CActionData object referenced by callback keys in the callback array of each registered animation set.

It should be noted that while the CActor destructor simply wraps a call to the Release method, this method is also called from other places. For example, it is called by the actor prior to loading an X file to make sure that all existing data is released prior to the new data being loaded.

CActor::ReleaseCallbackData

There are two versions of this function in CActor (i.e., it is overloaded). This version of the ReleaseCallbackData method takes no parameters and is called from the CActor::Release method discussed above. It loops through each animation set currently registered with the controller and fetches an interface to that animation set using the ID3DXAnimationController::GetAnimationSet method. Once we have a pointer to the animation set interface, we call the overloaded version of this function ReleaseCallbackData, passing in the animation set interface as a parameter. This overloaded version of the function (which we will discuss in a moment) does the work of releasing the callback key context data contained in the animation set. The code to the version of the function that is called by CActor::Release is shown below.

```
void CActor::ReleaseCallbackData( )
{
    HRESULT             hRet;
    ULONG               i, SetCount;
    ID3DXAnimationSet * pSet = NULL;

    if ( !m_pAnimController ) return;

    // For each animation set, release the callback data.
```

```
    for (i=0,SetCount=m_pAnimController->GetNumAnimationSets(); i<SetCount; ++i )
    {
        // Retrieve the animation set
        hRet = m_pAnimController->GetAnimationSet( i, &pSet );

        if ( FAILED(hRet) ) continue;

        // Release them for this set
        ReleaseCallbackData( pSet );

        // release the anim set interface
        pSet->Release();

    } // Next Animation Set
}
```

After we have passed the animation set interface into the overloaded version of the function, we can assume that any callback key context data associated with the current animation set has been released. As such, the animation set interface can now safely be released. We do this for all animation sets registered with the animation controller so that on function return all callback context data associated with the actor has been cleaned up.

Keep in mind that this function does not release any animation sets from memory. The animation sets are unregistered later in the CActor::Release method when the animation controller is destroyed. All we are doing here is giving the actor a chance to remove CActionData objects associated with callback keys from memory.

CActor::ReleaseCallbackData (Overloaded Method)

This version of the method is called by the function previously discussed. It is passed the interface of an ID3DXAnimationSet derived class which, in our lab project, is a CAnimationSet interface since we replaced all ID3DXKeyframedAnimationSets with our own implementation. This function will release callback data using only the methods of the ID3DXAnimationSet interface, so it will work correctly with any animation set derived from that interface.

This function is tasked with looping through each callback key registered with the passed animation set and releasing its context data (if it exists). The actor is not concerned with what the context data pointer of a callback key is pointing at, only that it is pointing at some object derived from IUnknown. Since the actor makes this assumption, it can simply loop through each callback key in the array and, if the context data pointer of the current key is non-NULL, cast the void pointer to an IUnknown interface and call its Release method.

We discussed the implementation of our CActionData class earlier and saw how its release method decremented the internal reference count and deleted the object (itself) from memory when the count hit zero. We also examined how the CActionData object is created in the callback key callback function and that when the CActionData object is returned to the actor by the callback function, the actor does not increment its reference count. Our application does not store any copies of this CActionData interface or increment its reference count anywhere else in the application. Each CActionData object will have a

reference count of 1 when first registered with the actor. Thus, when the actor calls the release method for each of its CActionData objects, the objects will delete themselves from memory.

This method works properly with any context data that is an IUnknown derived class. Therefore, you do not have to use our CActionData class; you can use your own context data objects and custom classes if you wish. Just make sure that they are derived from IUnknown and that you correctly implement the methods of the IUnknown interface.

Here is the code to the function that releases all context data for the callback keys registered with a single animation set.

```
void CActor::ReleaseCallbackData( ID3DXAnimationSet * pSet )
{
    double     Position  = 0.0;
    ULONG      Flags     = 0;
    IUnknown * pCallback = NULL;

    // Keep searching until we run out of callbacks
    for ( ;SUCCEEDED( pSet->GetCallback( Position,
                                         Flags,
                                         &Position,
                                         (void**)&pCallback )) ; )
    {
        // Ensure we only get the first available (GetCallback will loop forever).
        if ( Position >= pSet->GetPeriod() ) break;

        // Release the callback (it is required that this be an
        // instance of an IUnknown derived class)
        if ( pCallback ) pCallback->Release();

        // Exclude this position from now on
        Flags = D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION;

    } // Next Callback
}
```

The function loops through each callback key in the animation sets callback key array using the ID3DXAnimationSet::GetCallback method. The first time this method is called we include the initial position in the search (position 0.0). The function returns the track position and callback key context pointer for the next scheduled callback key.

When we covered the GetCallback method earlier, we learned that it will continue to loop when searching for keys. For example, if the period of the animation set was 10 seconds and the last callback key returned was the one at 10 seconds, the next time the function is called it would wrap around to the beginning of the array and return the first callback key again. We do not want this behavior in this case because we only need a single pass through the callback key array to release the context data. Thus we break from the loop as soon as the position variable is greater than the period of the animation. You can see that this position variable starts at zero, and is passed into each call to GetCallback method so that its value is overwritten by the position of the next callback key that is scheduled. As soon as this position reaches the length of the animation set (remember that the function returns a position analogous to a

non-looping track time), we know that we have processed all the keys and released any context data that needs releasing.

If we have not yet reached the period of the animation set, then we have a context data pointer returned for the next key in the list. If this is not NULL then it means that it must point to an object derived from IUnknown (our CActionData object for example). We simply cast this void pointer to an IUnknown interface pointer and call its release method. This will decrement the object reference count and cause it to delete itself from memory (unless you have incremented its reference count and stored pointers to these objects somewhere else in your application).

Finally, notice that after the first call to GetCallback is issued, for all future calls we pass in the `D3DXCALLBACK_SEARCH_EXCLUDING_INITIAL_POSITION` flag so that we do not get back the callback data we have just processed.

CActor::SetActorLimits

The application should have the ability to change the maximum limits of the actor's underlying controller both before and after the animation data has been loaded from an X file. This allows it to load an X file and then increase properties like the available mixer tracks from 2 (the default) to however many are desired.

Since this function can be called prior to or after the application has called the CActor::LoadActorFromX function, it works differently depending on whether or not the actor's animation controller already exists. If the controller already exists, then the new requested maximums are stored in the actor's member variables and are compared against the limits of the current controller. If any of the requested maximums are greater than what the current controller can handle, then this function will clone the animation controller into one that can support *at least* the requested maximums. The words 'at least' are important here because this function will only increase the limits of its internal animation controller; it will never decrease them.

If the application passes in zero for any of the properties, the maximum extents of the current controller for that property will be inherited from the previous controller by the clone. For example, if we pass in a MaxTracks parameter of zero but the controller already supports two, the cloned controller will support two tracks also. What is important is the fact that the while passed maximum extents are stored in the actor's member variables, the controller will only need to be cloned if one or more of the requested extents is greater than what is currently supported by the actor's animation controller.

The other mode of this function handles the case when the controller does not currently exist. This can happen if the application has called this function prior to calling LoadActorFromX. When this is the case, the maximum extents will simply be stored in the actor's member variables and the function will return. When the application later calls LoadActorFromX, the function will clone the animation controller automatically if the animation controller returned to the actor from the D3DXLoadMeshHierarchyFromX function does not support the previously set maximum extents the application desires.

The exception to the parameter list is the final parameter, MaxCallbackKeys. This value has nothing to do with the animation controller; it sets the size of the temporary callback key buffer that will be allocated in LoadActorFromX and passed to the application-defined callback key callback function. MaxCallbackKeys should be at least as large as the maximum number of callback keys your application intends to register with a single animation set. The default value is 1024, which was set in the constructor.

Let us now have a look at this function a section at a time.

```
HRESULT CActor::SetActorLimits(     ULONG MaxAnimSets /*= 0*/,
                                    ULONG MaxTracks /* = 0*/,
                                    ULONG MaxAnimOutputs /* = 0 */,
                                    ULONG MaxEvents /* = 0 */,
                                    ULONG MaxCallbackKeys /* = 0 */ )
{
    HRESULT hRet;
    bool    bCloneController = false;
    ULONG   Value;

    // Cache the current details
    if ( MaxAnimSets     > 0 ) m_nMaxAnimSets     = MaxAnimSets;
    if ( MaxTracks       > 0 ) m_nMaxTracks       = MaxTracks;
    if ( MaxAnimOutputs  > 0 ) m_nMaxAnimOutputs  = MaxAnimOutputs;
    if ( MaxEvents       > 0 ) m_nMaxEvents       = MaxEvents;
    if ( MaxCallbackKeys > 0 ) m_nMaxCallbackKeys = MaxCallbackKeys;

    // If we have no controller yet, we'll take this no further
    if ( !m_pAnimController ) return D3D_OK;
```

In this first section we test the passed parameters and if any are larger than zero, we take this as a request that the caller wants to change this property. Therefore, any parameter that is larger than zero is stored directly in the actor's member variables. If the controller is not yet created, then our work is done and we simply return. This will typically be all this function does when it is called from the application prior to the X file being loaded into the actor (or if the X file has no animation data, in which case, no controller will exist for the actor anyway).

If the controller is already created then we need to find out if we need to clone it. The first thing we do is copy the current maximum extents of the actor stored in its member variables into some local variables that we can resize later in the function. We reuse the parameter created variables for this purpose since the values passed in have either been stored in the actor (in which case we are just copying them straight back over again), or were zero (in which case we are copying over the previous limit the actor had set for that property). After this copy, we have the exact desired limits that this actor should be able to support in local variables that we can manipulate.

```
    // Store back in our temp variables for defaults
    MaxAnimSets     = m_nMaxAnimSets;
    MaxTracks       = m_nMaxTracks;
    MaxAnimOutputs  = m_nMaxAnimOutputs;
    MaxEvents       = m_nMaxEvents;
    MaxCallbackKeys = m_nMaxCallbackKeys; // Should never be zero
```

When we clone the animation controller, we must pass all the limits of the new controller into the cloning function; not just the ones we wish to change. This makes sure that all limits are inherited by the cloned controller. If the actor has a controller that was previously limited to two mixing tracks, we might call this method and simply pass in zero as this limit. This states that we are quite happy with the current limit and wish it to survive the clone. The local variables in the above code now contain the limits for each controller property that we would like in our cloned controller.

However, as mentioned earlier in this workbook, we only wish to clone the controller if the desired limits exceed those of the pre-existing controller. If the current controller already has limits that are equal to or exceed the requested limits, no clone takes place. The current controller will be able to handle the applications requirements in that case.

The next section of code extracts each limit from the current controller and tests to see if it is smaller than the requested limit. If it is, then the controller will need to be cloned, so we set the bCloneController Boolean to true. If the requested limit is less than or equal to the current limit of the controller, we simply set the local limit variable to the current limit of the controller.

```
    // Otherwise we must clone if any of the details are below our thresholds
    Value = m_pAnimController->GetMaxNumAnimationOutputs();

    if ( Value < MaxAnimOutputs )
        bCloneController = true; else MaxAnimOutputs = Value;

    Value = m_pAnimController->GetMaxNumAnimationSets();

    if ( Value < MaxAnimSets )
        bCloneController = true; else MaxAnimSets = Value;

    Value = m_pAnimController->GetMaxNumTracks();
    if ( Value < MaxTracks )
         bCloneController = true; else MaxTracks = Value;

    Value = m_pAnimController->GetMaxNumEvents();
    if ( Value < MaxEvents )
        bCloneController = true; else MaxEvents = Value;
```

To understand why this code is necessary, let us imagine that we have a controller that is currently capable of managing 100 animation outputs. Let us also imagine that the application calls this method and specifies an animation output limit of 80. In this case, the controller does not need to be cloned because we only wish to expand the limits of the controller, not contract them. We made this design decision to prevent the application from accidentally breaking the animation data loaded from the X file. In this example, the actor's requested limit (80) would be compared against the current limit of the controller (100) and the Boolean is not set to true. Instead we simply upgrade the requested limit (MaxAnimOutputs=80) to the current limit of the controller (MaxAnimOutputs=100).

While this might seem like a strange thing to do, remember that while this particular limit might not cause the controller to be cloned, one of the other limits of the controller may be lower than requested, causing the controller to be cloned anyway. By upgrading the local variable limits so that they are at least as large as the current controller, that limit will be inherited by the clone. If the MaxAnimSets

member contained a value of 10 but the current controller only supported 2 tracks, the controller would be cloned. However, when we pass the MaxAnimOutputs local variable into the clone function, we are specifying that the controller be created with 100 animation outputs (the original maximum) instead of 80 animation outputs (the requested amount). So this code is necessary because we want to make sure that the limits of the original controller that will not change survive in the cloned controller. If we did not have this safeguard, and we cloned the animation controller with 80 animation outputs instead of 100, the animation data for 20 frames in the hierarchy would be lost in the clone. This is almost always something that the application would not intend to happen.

At this point in the code, our Boolean variable will indicate that the controller either does or does not need cloning. If it does then we pass our local variables as the limits to the ID3DXAnimationController::CloneAnimationController method to create a new controller with the desired limits. All animation data that existed in the original controller will now be present in the clone (even our context data pointers).

Once the clone has been performed, we release the original controller's interface and assign the actor's m_pAnimController pointer to the newly cloned interface. The remaining section of the code that performs the clone is shown below.

```
    // Clone our animation controller if there are not enough set slots available
    if ( bCloneController )
    {
        LPD3DXANIMATIONCONTROLLER pNewController = NULL;

        // Clone the animation controller
        hRet = m_pAnimController->CloneAnimationController( MaxAnimOutputs,
                                                            MaxAnimSets,
                                                            MaxTracks,
                                                             MaxEvents,
                                                           &pNewController );
        if ( FAILED(hRet) ) return hRet;

        // Release our old controller
        m_pAnimController->Release();

        // Store the new controller
        m_pAnimController = pNewController;

    } // End if requires clone

    // Success
    return D3D_OK;
}
```

CActor::LoadActorFromX

The LoadActorFromX function is another method that existed in the previous lab project but has now been upgraded to facilitate the incorporation of animation data. The code that has been added is highlighted in bold. The changes may look trivial but this is because the method performs most of the new functionality in new member functions of the actor. Let us look at it a bit at a time.

```
HRESULT CActor::LoadActorFromX( LPCTSTR FileName, ULONG Options,
                                LPDIRECT3DDEVICE9 pD3DDevice,
                                  bool bApplyCustomSets /* = true */ )
{
    HRESULT hRet;
    CAllocateHierarchy Allocator( this );

    // Validate parameters
    if ( !FileName || !pD3DDevice ) return D3DERR_INVALIDCALL;

    // Release previous data!
    Release();

    // Store the D3D Device here
    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();

    // Store options
    m_nOptions = Options;

    // Load the mesh heirarchy and the animation data etc.
    hRet = D3DXLoadMeshHierarchyFromX( FileName, Options, pD3DDevice, &Allocator,
                                       NULL, &m_pFrameRoot, &m_pAnimController );
    if ( FAILED(hRet) ) return hRet;

    // Copy the filename over
    _tcscpy( m_strActorName, FileName );
```

This first section of the code is almost entirely unchanged. It simply increases the reference count of the passed Direct3D device and stores its pointer in a member variable. It then uses the D3DXLoadMeshHierarchyFromX function to load the frame hierarchy. The only update here is that we now pass the address of an ID3DXAnimationController interface pointer to the loading function. If the X file contains animation data then an animation controller will be created for us along with its animation sets and it will be returned to us via this pointer. At this point, we will have a pointer to the animation controller interface for our hierarchy stored in the actor's m_pAnimationController member variable.

We then copy the name of the X file from which the actor was loaded into the m_strActorName member variable. We saw in the previous lab project that the application uses the actor's name when processing X file references inside an IWF file. If an X file has already been loaded, then an actor with the same name as the X file will already exist in the scene. The application can then choose to instance that actor instead. Now however, the actor name also plays another role. We discussed earlier that the application defined callback function for feeding in callback keys to the actor will also be passed the name of the

actor. This allows the application defined callback to test whether this is an X file it wishes to define callback keys for.

The next section of code is new. The first thing the actor does once it knows it has a valid controller is call the CActor::SetActorLimits method:

```
    // Apply our derived animation sets
    if ( m_pAnimController )
    {
        // Apply our default limits if they were set before we loaded
        SetActorLimits( );
```

This time, the function is passed no parameters, indicating that the parameters passed into the function will all be zero by default. When we discussed this function earlier we learned that when a given limit is zero, the current value for that limit stored in the actor's member variables is used. If the application has used the SetActorLimits method to set its desired limits for the controller prior to calling the LoadActorFromX function, the clone will be performed if the controller created for us by D3DX is not capable of fulfilling our application requirements.

The next line of code is very important. It tests to see if a callback key callback function has been registered with this actor by the application. If it has, then the pFunction pointer of the CALLBACK_FUNC structure in the fourth element (CALLBACK_CALLBACKKEYS = 4) of the actor's callback function array will not be NULL. If this is the case, a new method, which we have not yet discussed, is called. This method is called ApplyCallbacks and will be discussed next.

The ApplyCallbacks function will loop through every animation set currently registered with the controller, and for each one, will call the application defined callback function. The callback function will be passed a temporary array of D3DXKEY_CALLBACK structures. The application can then store callback keys and context data in this array before returning it back to the ApplyCallback method. This method will then replace the current animation set with a new copy of the animation set that has the callback keys registered with it. Remember, we have to do this because we have no way of registering callback keys with an animation set after it has been created.

```
        // If there is a callback key function registered, collect the callbacks
        if ( m_CallBack[CALLBACK_CALLBACKKEYS].pFunction != NULL )
             ApplyCallbacks();
```

We will look at the ApplyCallbacks method in a moment, but just note that when it returns, our animation controller may now contain animation sets that have callback keys registered with them.

The final section of the function code uses another new CActor method called ApplyCustomSets. As discussed earlier in this workbook, at the time of writing, a bug existed in the ID3DXKeyframedAnimationSet::GetSRT method. Because of this, we constructed a way around the problem by replacing all ID3DXKeyframedAnimationSets registered with our controller with CAnimationSet copies. That is what the ApplyCustomSets method does (we will cover the code to this method in a moment). It loops through every animation set currently registered with the controller and creates a replacement CAnimationSet object for each. It then copies over all the animation data from the

original keyframed animation set into our own CAnimationSet. Once we have a copy of each animation set, we release the original keyframed animation sets and register our CAnimationSet objects in their place. We covered all of the code for CAnimationSet earlier in this workbook.

```
        // Apply the custom animation set classes if we are requested to do so.
        if ( bApplyCustomSets == true )
        {
            hRet = ApplyCustomSets();
            if ( FAILED(hRet) ) return hRet;

        } // End if swap sets

    } // End if any animation data

    // Success!!
    return D3D_OK;
}
```

Notice that the LoadActorFromX method accepts a Boolean parameter called bApplyCustomSets. Only if this value is set to true will the ApplyCustomSets method be called and our own CAnimationSet implementations replace the ID3DXKeyframedAnimationSets registered with the controller. This parameter allows us to quickly unplug our CAnimationSet objects from the system when the bug in ID3DXKeyframedAnimationSet is fixed in the future.

While only a small amount of code was added to LoadActorFromX, there were calls made to two new methods which we will look at next. These two new methods do contain significant amounts of code.

CActor::ApplyCallbacks

The ApplyCallbacks method is called by LoadActorFromX after the actor has been loaded and already has an animation controller. The purpose of this function is to provide the application with a means of registering callback keys for animation sets registered with the animation controller. Keys cannot be registered with an animation set after it has been created, so we provide this functionality by making new copies of the animation sets for which the application has specified callback keys. We can register these new callback keys as we create the new animation set copies and then release the original animation sets and register the new ones (with the callback keys) in their place.

The first section of this function is shown next. It checks that the animation controller of the actor is valid and that it already has registered animation sets. If not, then the function simply returns. Next, we store (in the SetCount local variable) the number of animation sets currently registered with the animation controller. We will need to loop through each animation set and call the application defined callback function (if it has been registered) to give the application a chance to specify callback keys for each animation set.

Once we have recorded the number of sets in the animation controller, we allocate a temporary array of D3DXKEY_CALLBACK structures. For each animation set processed, we will pass this array into the application defined callback function. The callback function can then fill this array with callback data as we saw earlier in this workbook. Notice that the size of this temporary array is determined by the actor limit m_nMaxCallbackKeys, which can be changed using the SetActorLimits method. This is initially

set to 1024, which means the callback function will have enough room to register 1024 callback keys in this array for a single animation set. Remember, this is a per-set limit since the temporary buffer will be reused for each animation set processed by this function. The m_nMaxCallbackKeys member variable allows us to control the temporary memory overhead used by this function.

```
HRESULT CActor::ApplyCallbacks( )
{
    ULONG   i, j;
    HRESULT hRet;
    ID3DXKeyframedAnimationSet * pNewAnimSet = NULL, **ppSetList = NULL,
                               * pKeySet = NULL;
    ID3DXAnimationSet          * pAnimSet    = NULL;
    D3DXKEY_CALLBACK           * pCallbacks  = NULL;

    // Validate
    if ( !m_pAnimController || !m_pAnimController->GetNumAnimationSets( ) )
         return D3DERR_INVALIDCALL;

    // Store set count
    ULONG SetCount = m_pAnimController->GetNumAnimationSets( );

    // Allocate our temporary key buffer
    pCallbacks = new D3DXKEY_CALLBACK[ m_nMaxCallbackKeys ];
    if ( !pCallbacks ) return E_OUTOFMEMORY;

    // Allocate temporary storage for our new animation sets here
    ppSetList = new ID3DXKeyAnimationSet*[ SetCount ];

    if ( !ppSetList ) { delete pCallbacks; return E_OUTOFMEMORY; }
```

Once we have allocated the callback key array, we allocate an array of ID3DXKeyAnimationSet interface pointers. The size of this array is equal to the number of animation sets currently registered with the controller.

We have several things we must consider when implementing this function. First, we will need to unregister any animation sets that need to be replaced. These will only be animation sets that have a callback key count greater than zero. For any others we will let them remain registered with the controller. Second, we may have a situation where the controller has some animation sets that are not of the ID3DXKeyframedAnimationSet type (e.g., custom sets, compressed animation sets, etc.). Since we can only register callback keys with keyframed animation sets, we will want to also leave these others alone, registered with the controller. To ease this process, we have decided to implement this in three passes/loops. This makes the code slightly longer but easier to understand and maintain.

In the first loop, we will fetch each animation set from the animation controller and store a copy of its interface pointer in the temporary ID3DXAnimationSet array we have just allocated. In the loop, we will call ID3DXAnimationController::GetAnimationSet to fetch each set. As this function returns a pointer to the base interface, this is ideal. We simply store pointers to all the animation sets (keyframed or otherwise) in our temporary animation set array and then remove that set from the controller. What we will have at the end of this first pass is an empty animation controller, with pointers to each animation set it used to have registered in our temporary array. We can then work directly with this array, rejecting

or replacing animation sets we no longer want and leaving the ones we do want unaltered. At the end of the function, we can loop through this array and register all the animation sets back with the controller. Some of these animation sets will have been replaced by copies containing callback key data.

Here is the code that performs this first un-register pass.

```
for ( i = 0; i < SetCount; ++i )
    {
        // Keep retrieving index 0 (remember we unregister each time)
        hRet = m_pAnimController->GetAnimationSet( 0, &pAnimSet );
        if ( FAILED( hRet ) ) continue;

        // Store this set in our list, and AddRef
        ppSetList[ i ] = pAnimSet;
        pAnimSet->AddRef();

        // Unregister the old animation sets first otherwise we won't have enough
        // 'slots' for the new ones.
        m_pAnimController->UnregisterAnimationSet( pAnimSet );

        // Release this set, leaving only the item in the set list.
        pAnimSet->Release();

    } // Next Set
```

Notice that when we copy the interface pointer for each animation set, we increase its reference count. If you study the previous code, you will realize that prior to the GetAnimationSet call, the animation controller is the only object that has an interface to each animation set, so the reference count of the animation set would be 1. When we call GetAnimationSet, the controller returns an interface to our application, automatically increasing the reference count before returning the pointer. The reference count of the animation set at this point is 2. We then copy this interface pointer into our array and increase the reference count again, so the reference count is 3. This is correct since there are three pointers to this interface currently in use. The animation controller has its own pointer to it, the pAnimSet local variable currently has a reference to it, and our interface pointer array has the third. We then call the animation controller's UnregisterAnimatonSet method so that the animation set releases its pointer to the interface, taking the reference count down to 2. Next we release the temporary pAnimSet pointer which decrements the reference count to 1. This is now exactly how we want it -- the only reference to this animation set is now in our local ID3DXAnimationSet array.

The next pass through the animation sets is where all the work is done. We loop through each animation set in our temporary array and query its interface. If it is not an ID3DXKeyframedAnimationSet interface, we will skip to the next iteration of the loop. We are essentially leaving this animation set alone and will re-add it to the controller later in the function. It will be unaltered from its initial state. If it is a keyframed animation set, then the QueryInterface method will return us a pointer to that interface:

```
    // Loop through each animation set and rebuild a new one containing the
    // callbacks.
    for ( i = 0; i < SetCount; ++i )
    {
```

```
        // Skip if we didn't manage to retrieve
        if ( ppSetList[i] == NULL ) continue;

        // Retrieve the set we're working on
        pAnimSet = ppSetList[i];

        // Query the interface to get a keyframed animation set
        hRet = pAnimSet->QueryInterface( IID_ID3DXKeyframedAnimationSet,
                                         (void**)&pKeySet );
        if ( FAILED(hRet) ) continue;
```

If we get this far in this iteration, we have a keyframed animation set which the application may wish to register callback keys for. We now have to call the application defined callback function to see if that is the case.

In the following code we see a technique we have used many times before when calling callback functions. If the application has registered a callback key callback function with the actor, its function pointer will exist in the fourth element in the actor's callback function array. If you examine CActor.h you will see that we have defined a type called COLLECTCALLBACKS which describes a pointer to this method:

```
typedef ULONG (*COLLECTCALLBACKS )( LPVOID pContext,
                                    LPCTSTR strActorFile,
                                    LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                    D3DXKEY_CALLBACK pKeys[] );
```

As you can see, any variables we declare with this type will be pointers to functions with the exact function signature of our application defined callback function (dictated by the actor). Your callback function *must* use this exact signature.

In the following code we create a pointer of this type called CollectCallbacks and assign it the value of the pFunction member of the fourth element in the actor's callback array. If the application has registered a callback function of this type with the actor (using the CActor::RegisterCallback method), then we will now have a pointer to the function that we can call. Otherwise, this element will be set to NULL, indicating the application has expressed no desire to register any callback keys with this actor's animation sets.

> **Note**: It is important to remember that the application must register any callback functions with the actor prior to calling LoadActorFromX because these functions are called in the middle of the loading process. This was also true of the texture and material callback functions that were covered in the previous lab projects.

In the next section of code, we fetch the function pointer and use it to invoke the callback function, passing any context data that was registered (the CScene object instance in our project). We also pass the function the name of the actor so that the callback function can determine which actor is making the call and, therefore, what callback data that actor should receive. We also pass in a pointer to the animation set interface so the callback function can extract information about which set it is being called for. In the example callback function we showed earlier, the animation set's interface was used to extract the name of the set. This allowed the callback function to determine what callback data should be

registered (if any) with the current animation set being processed. Finally, we pass in a pointer to our temporary D3DXKEY_CALLBACK array so that the application can fill it with callback data for the current animation set.

When the callback function returns, it will return an integer describing how many keys were placed in the array by the callback function. If the function returns zero, then we do not have to alter this animation set; we can simply continue to the next iteration of the loop and leave this animation set alone. This animation set will be re-registered with the animation controller later in the function. Here is the code that shows the process just described.

```
        // Retrieve our application's registered function pointer
        COLLECTCALLBACKS CollectCallbacks
            = (COLLECTCALLBACKS)m_CallBack[ CALLBACK_CALLBACKKEYS ].pFunction;

        // Request that our key buffer be filled
        ULONG KeyCount
           = CollectCallbacks( m_CallBack[ CALLBACK_CALLBACKKEYS ].pContext,
                               m_strActorName, pKeySet, pCallbacks );

        // If no keys are to be added, leave the existing one alone.
        if ( KeyCount == 0 ) { pKeySet->Release(); continue; }
```

If we get this far, then the current animation set we are processing is one that the application has supplied us with callback keys for. Therefore, as callback keys can only be registered with an animation set at creation time, we need to create a new copy of this animation set which will eventually be used to replace it. In the next section of code we create a new animation set that has the same name, source ticks per second, playback type, and number of animation as the animation set we are copying.

```
        // Create a new animation set for us to copy data into
        hRet = D3DXCreateKeyframedAnimationSet( pKeySet->GetName(),
                                                pKeySet->GetSourceTicksPerSecond(),
                                                pKeySet->GetPlaybackType(),
                                                pKeySet->GetNumAnimations(),
                                                KeyCount,
                                                pCallbacks,
                                                &pNewAnimSet );

        if ( FAILED(hRet) ) { pKeySet->Release(); continue; }
```

We now have a new animation set which has the same parameters as the original animation set, with one big difference. Notice that when we call the D3DXCreateKeyframedAnimationSet method, now we also pass in the number of callback keys (returned by the callback function) and a pointer to the array containing those keys (populated by the callback function). As result, the new animation set created will contain the callback key data supplied by the callback function.

Of course our job is not yet done. Although our new animation set shares the same properties as the original set and now includes callback data, it still contains no animation data. Although our new animation set has had space reserved for the correct number of animations, these animations have no data. We will have to manually extract the SRT keyframes from each animation in the original set and assign them to matching animations in the new set.

The following code performs the copying of the SRT data for each animation. It sets up a loop to iterate through each animation in the original set. For each animation, it first fetches the number of scale, rotate, and translate keys registered with that animation. These three values are then used to allocate temporary arrays to store copies of the SRT data for this animation. We then copy the scale, rotation and translation keys from the original animation into these three arrays.

```
        // Copy over the data from the old animation set
        //(we've already stored the keys)
        for ( j = 0; j < pKeySet->GetNumAnimations(); ++j )
        {
            LPCTSTR               strName = NULL;
            ULONG                 ScaleKeyCount, RotateKeyCount,
                                  TranslateKeyCount;
            D3DXKEY_VECTOR3     * pScaleKeys = NULL, * pTranslateKeys = NULL;
            D3DXKEY_QUATERNION  * pRotateKeys = NULL;

            // Get the old animation sets details
            ScaleKeyCount     = pKeySet->GetNumScaleKeys( j );
            RotateKeyCount    = pKeySet->GetNumRotationKeys( j );
            TranslateKeyCount = pKeySet->GetNumTranslationKeys( j );

            pKeySet->GetAnimationNameByIndex( j, &strName );

            // Allocate any memory required
            if ( ScaleKeyCount     )
                pScaleKeys     = new D3DXKEY_VECTOR3[ ScaleKeyCount ];

            if ( RotateKeyCount    )
               pRotateKeys    = new D3DXKEY_QUATERNION[ RotateKeyCount ];

            if ( TranslateKeyCount )
               pTranslateKeys = new D3DXKEY_VECTOR3[ TranslateKeyCount ];
            // Retrieve the keys
            if ( pScaleKeys     ) pKeySet->GetScaleKeys( j, pScaleKeys );
            if ( pRotateKeys    ) pKeySet->GetRotationKeys( j, pRotateKeys );
            if ( pTranslateKeys ) pKeySet->GetTranslationKeys( j, pTranslateKeys );
```

We now have the SRT data for the current animation we are copying over. Our next task is to register these SRT keys with the matching animation in the new animation set using the ID3DXKeyframedAnimationSet::RegisterAnimationSRTKeys method. When this function returns, D3DX will have copied the SRT data into the animation in the new set. These temporary SRT arrays are no longer needed and can be deleted.

```
            hRet = pNewAnimSet->RegisterAnimationSRTKeys(  strName,
                                                           ScaleKeyCount,
                                                           RotateKeyCount,
                                                           TranslateKeyCount,
                                                           pScaleKeys,
                                                           pRotateKeys,
                                                           pTranslateKeys, NULL );
            // Delete the temporary key buffers
            if ( pScaleKeys     ) delete []pScaleKeys;
            if ( pRotateKeys    ) delete []pRotateKeys;
```

```
            if ( pTranslateKeys ) delete []pTranslateKeys;

            // If we failed, break out
            if ( FAILED(hRet) ) break;

        } // Next Animation 'item'
```

If for some reason we got this far but were forced to break out of the above loop before each animation had been copied, then it means something went wrong during the copy procedure. The next section of code shows that if this is the case, we will release the new animation set and leave the original animation as is. We just continue to the next iteration of the loop to process the next animation set. This is a safe way to try to continue the process in the event of some unforeseen error.

```
        // If we didn't reach the end, this is a failure
        if ( j != pKeySet->GetNumAnimations() )
        {
            // Release new set, and reference to old
            pKeySet->Release();
            pNewAnimSet->Release();
            continue;

        } // End if failed
```

At this point we have copied over all the animation data from the original animation set into the new animation. The original animation set can now be released from memory as it will no longer be needed. Notice how we perform two releases to free the original animation set. The pKeySet pointer is the ID3DXKeyframedAnimationSet interface to the original animation set that we retrieved with the QueryInterface call. We then release the original animation set interface pointer stored in thre array and replace it with our new copy.

```
        // Release our duplicated set pointer
        pKeySet->Release();

        // Release the actual animation set, memory will now be freed
        pSetList[I]->Release();

        // Store the new animation set in our temporary array
        ppSetList[i] = pNewAnimSet;

    } // Next Set
```

Now every animation set has been processed. Some may have been replaced with copies, while others may have been left in their original state. Either way, at this point, the ppSetList array contains all the animation sets we wish to register with the animation controller. In the third pass we do exactly that. We loop through each animation set in this array and register it with the animation controller. We then release the interface pointer stored in the array since we no longer need it. This will not cause the animation set to be freed from memory because the animation controller has a pointer to it now and will have incremented its reference count when the set was registered.

```
    // third pass, register all new animation sets
```

```
    for ( i = 0; i < SetCount; ++i )
    {
        // Nothing stored here?
        if ( !ppSetList[i] ) continue;

        // Register our brand new set. (AddRef is called)
        m_pAnimController->RegisterAnimationSet( ppSetList[i] );

        // Release our hold on the custom animation set
        ppSetList[i]->Release();

    } // Next Set
```

The last section of the function frees the temporary animation set array and callback key array from memory. It also assigns the controller's first animation set to track zero by default. If we did not do this then no animation set would be assigned to any track. Remember, although the D3DXLoadMeshHierarchyFromX function automatically assigns an animation set to track zero on our behalf, we have unregistered all of those animation sets and undone this mapping.

```
    // Free up any temporary memory
    delete []ppSetList;
    delete []pCallbacks;

    // Set the first animation set into the first track, otherwise
    // nothing will play, since all previous tracks have potentially been
    //  unregistered.
    SetTrackAnimationSetByIndex( 0, 0 );

    // We're all done
    return D3D_OK;
}
```

That was a sizable function but ultimately very straightforward. We are essentially just cloning animation sets manually so that we can add additional data to them.

CActor::ApplyCustomSets

One of the last methods to be called from the LoadActorFromX method is CActor::ApplyCustomSets. This function gives us a chance to replace all ID3DXKeyframedAnimationSets with our own CAnimationSet objects. You will recall from our earlier discussion that this allows us to sidestep a bug in the ID3DXKeyframedAnimationSet::GetSRT method that existed at the time of this writing.

This method follows, for the most part, the same logic as the CActor::ApplyCallbacks method just discussed. In fact, it executes using the same three pass apporach. First it copies all the animation sets from the controller into a locally allocated array. Each animation set is then un-registered with the controller. In the second pass we loop through each animation set in the array. For each ID3DXKeyframedAnimationSet we find, we copy its information into a new CAnimationSet object. Luckily, the CAnimationSet constructor takes care of the actual copying of the keyframe data from the passed ID3DXKeyframedAnimationSet, so this code is a lot shorter than the previous function. For each

set we clone, we release the original. As before, if we find any compressed or custom animation sets we leave them alone in the array for re-registration later in the function. At the end of the second pass, we will have an array of animation sets where each keyframed animation set has been replaced by a CAnimationSet object. In the third pass, we simply register every animation set in this array with the animation controller.

Note however that this method is called by CActor::LoadActorFromX after the ApplyCallbacks method has been called. Therefore, the animation sets currently registered with the controller may well have callback keys defined for them. As we saw when we covered the CAnimationSet constructor, the callback keys of the animation set are also copied over into the new CAnimationSet copy.

The first pass of this function should look familiar. It allocates an array of ID3DXAnimationSet interfaces large enough to store a copy of each animation set currently registered with the controller.

```
HRESULT CActor::ApplyCustomSets( )
{
    ULONG   i;
    HRESULT hRet;
    CAnimationSet       * pNewAnimSet = NULL;
    ID3DXAnimationSet   * pAnimSet    = NULL, ** ppSetList = NULL;

    // Validate
    if ( !m_pAnimController || !m_pAnimController->GetNumAnimationSets( ) )
        return D3DERR_INVALIDCALL;

    // Store set count
    ULONG SetCount = m_pAnimController->GetNumAnimationSets( );

    // Allocate array to hold animation sets
    ppSetList = new ID3DXAnimationSet*[ SetCount ];
    if ( !ppSetList ) return E_OUTOFMEMORY;

    // Clear the array
    ZeroMemory( ppSetList, SetCount * sizeof(ID3DXAnimationSet*) );
```

We now loop and fetch each animation set from the animation controller and store the retrieved base interface pointer in our local animation set array. As we copy over each animation set interface, we release that animation set from the controller so that the only reference to the animation set interface is now in our array.

```
    // Loop through each animation set and copy into our set list array
    for ( i = 0; i < SetCount; ++i )
    {
        // Keep retrieving index 0 (remember we unregister each time)
        hRet = m_pAnimController->GetAnimationSet( 0, &pAnimSet );
        if ( FAILED( hRet ) ) continue;

        // Store this set in our list, and AddRef
        ppSetList[ i ] = pAnimSet;
        pAnimSet->AddRef();
```

```
        // Unregister the old animation sets first otherwise we won't have enough
        // 'slots' for the new ones.
        m_pAnimController->UnregisterAnimationSet( pAnimSet );

        // Release this set, leaving only the item in the set list.
        pAnimSet->Release();

    } // Next Set
```

The second pass is where we copy over the data from the original animation set into a new CAnimationSet. We do this by allocating a new CAnimationSet object and passing into the constructor the ID3DXAnimationSet base interface of the set we intend to clone. The constructor will take care of copying over all the keyframe data and storing it in the CAnimationSet's internal variables (if it is an ID3DXKeyframedAnimationSet that is passed in). Because this copying is happening in the constructor we have no way of returning success or failure, so we use exceptions. If an exception is thrown from inside the constructor, the catch block will simply delete the CAnimationSet we just allocated. When this happens, the original animation set's pointer is left in our array and will be re-registered with the controller later on.

```
    // Loop through each animation set and rebuild a new custom one.
    for ( i = 0; i < SetCount; ++i )
    {
        // Skip if we didn't retrieve anything
        if ( ppSetList[i] == NULL ) continue;

        // Allocate a new animation set, duplicating the one we just retreived
        // Note : Because this is done in the constructor, we catch the
        // exceptions because no return value is available.
        try
        {
            // Duplicate
            pNewAnimSet = new CAnimationSet( ppSetList[i] );

        } // End Try Block
        catch ( HRESULT & e )
        {
            // Release the new animation set
            delete pNewAnimSet;

            // Just skip this animation set (leave it in the array)
            continue;

        } // End Catch block
        catch ( ... )
        {
            // Just skip this animation set (leave it in the array)
            continue;

        } // Catch all other errors (inc. out of memory)
```

If we get this far without incident, then the animation set has been successfully copied into a new CAnimationSet object. The original animation set can then be released and a pointer to our new CAnimationSet stored in the array in its place.

```
        // Release this set, memory will now be freed
        ppSetList[i]->Release();

        // Store the new animation set in our temporary array
        ppSetList[i] = pNewAnimSet;

    } // Next Set
```

Finally, in the third pass we loop through each animation set in the array and register it with the controller. We then delete the temporary array and bind the first animation set to the first track of the controller.

```
    // third pass, register all new animation sets
    for ( i = 0; i < SetCount; ++i )
    {
        // Register our brand new set. (AddRef is called)
        m_pAnimController->RegisterAnimationSet( ppSetList[i] );

        // Release our hold on the custom animation set
        ppSetList[i]->Release();

    } // Next Set

    // Free up any temporary memory
    delete []ppSetList;

    // Set the first animation set into the first track, otherwise
    // nothing will play, since all previous tracks have been unregistered.
    SetTrackAnimationSetByIndex( 0, 0 );

    // We're all done
    return D3D_OK;

}
```

When this function returns back to LoadActorFromX, all ID3DXKeyframedAnimationSets will have been replaced with our CAnimationSets.

We have now covered all the new methods involved with loading and initializing the actor for animation. The remaining functions exposed by the actor are used by the application to play animations and configure the playback settings.

CActor::AdvanceTime

The new AdvanceTime method is the heartbeat of our actor. The application will call this method periodically to update the animation system's internal clock. This is really just a wrapper function around a call to the ID3DXAnimationController::AdvanceTime method, which updates the animation system and generates the new relative frame matrices for the hierarchy.

The first thing you will often want to do after updating the frame matrices is traverse the hierarchy and build the absolute matrices for each frame. You will recall from the previous lab project that we did this, during each iteration of our game loop, using the CActor::UpdateFrameMatrices method. This method traverses the frame hierarchy and combines all relative matrices to generate the world matrices for each frame in the hierarchy. It makes sense that we would want to perform this task *after* animating the local frame matrices because this step would invalidate the previous world matrices.

For convenience, we have added a Boolean parameter called UpdateFrames to this function (set to true by default). When set to true, the function will automatically call CActor::UpdateFrameMatrices before returning. This allows the actor to update the animation and rebuild the absolute matrices for each frame with a single call to CActor::AdvanceTime. The UpdateFrameMatrices method is unchanged from the previous lab project so will not be covered again here.

There may be times when you do not wish the world matrices of each frame to be calculated immediately after an animation update has taken place. In this case you can pass false as this parameter. You might do this because the application intends to make some manual adjustments to the relative frame matrices prior to the world matrix rebuilding. If the application does pass false to the UpdateFrames parameter, it must make sure that it calls CActor::UpdateFrameMatrices prior to rendering the actor. If it does not, the animations applied to the relative frame matrices will not be reflected in their world space counterparts used for the mesh transformations.

The CActor::AdvanceTime method is shown below.

```
void CActor::AdvanceTime(     double fTimeElapsed,
                              bool UpdateFrames /* = true */,
                              LPD3DXANIMATIONCALLBACKHANDLER pCallbackHandler)
{
    if ( !IsLoaded() ) return;

    // Set the current time if applicable
    if ( m_pAnimController )
        m_pAnimController->AdvanceTime( fTimeElapsed, pCallbackHandler );

    // Update the frame matrices
    if ( UpdateFrames ) UpdateFrameMatrices( m_pFrameRoot, &m_mtxWorld );
```

The application passes the amount of elapsed time since the previous call to the function, which is then routed to the ID3DXAnimationController::AdvanceTime method. As discussed earlier, when the ID3DXAnimationController::AdvanceTime method returns program flow back to our function, the relative hierarchy frame matrices will have been updated to represent their new positions. Finally, we call the UpdateFrameMatrices method which uses the updated relative matrices to build the

absolute/world matrices for each frame that will be used for transformation and rendering of any meshes in the hierarchy.

As the third parameter to the CActor::AdvanceTime method, the application can pass a pointer to a ID3DXAnimationCallbackHandler derived class. This base abstract interface exposes only one function which we must implement if we wish our callback keys to be used. When the controller executes a callback key, it will call the ID3DXAnimationCallbackHandler::HandleCallback method. Obviously this will only happen if we have passed into the controller the pointer to an object that is derived from this interface.

If you take a look in the d3dx9anim.h header file which ships with the SDK, you can see that the ID3DXAnimationCallbackHandler interface is declared as:

```
DECLARE_INTERFACE(ID3DXAnimationCallbackHandler)
{
    STDMETHOD(HandleCallback)(THIS_ UINT Track, LPVOID pCallbackData) PURE;
};
```

This is a pure virtual base class rather than an interface; it is not even derived from IUnknown. What does this mean for us? Well, all we have to do is derive our own class from this base class and implement its single method: HandleCallback. We can then pass a pointer to our object into the CActor::AdvanceTime method, where it will passed to the ID3DXAnimationController::AdvanceTime method. From there, the controller will call its HandleCallback method whenever a callback key is executed.

When this method is called by the controller, it is passed two parameters. The first is the track number for which the callback key was triggered. Your callback function might use this to determine if it wishes to take any action with respect to the current track. The second parameter is the callback key context data pointer. Remember, this will contain either NULL or the address of a CActionData object. Recall from our earlier discussion that our actor enforces the rule that callback key context pointers should point to an IUnknown derived class so that the actor can clean it up properly. In our application, we use the CActionData object for storing data that we would like passed to our callback method. In Lab Project 10.1, each CActionData object will contain the name of a wave file sound effect that the callback function can play when the key is triggered.

If you look in CScene.h you will see that we have derived our own class from ID3DXAnimationCallbackHandler like so:

```
class CActionHandler : public ID3DXAnimationCallbackHandler
{
public:
     STDMETHOD(HandleCallback)(THIS_ UINT Track, LPVOID pCallbackData);
};
```

Note that we have added nothing to this class and used a straight implementation of the interface laid out by the base class. It is a pointer to a CActionHandler class that we will pass into the

CActor::AdvanceTime method. Its HandleCallback method will be called by the controller whenever a callback key needs to be processed.

The CActionHandler::HandleCallback method is implemented in CScene.cpp (see below). Our callback function is very simple and does not use the track number passed by the controller for any purpose. It is only interested in the second parameter, which contains a pointer to our CActionData object (containing the name of a sound file that will need to be played).

The first thing we do is cast the void context data to an IUnknown interface. We know that whatever object may be pointed at by this void pointer has to be either NULL or one derived from IUnknown since our CActor class insists on it. Once we have cast it to IUnknown, we had better make sure that this is a CActionData object since it is what our callback handler is interested in handling. Therefore, we call IUnknown::QueryInterface to check that it is indeed one of our CActionData objects. If it is not, then we return immediately since this callback function does not know how to handle any other data type. If it is a CActionData object, then we are returned a CActionData interface which we can then use to extract the wave filename.

```
HRESULT CActionHandler::HandleCallback( UINT Track, LPVOID pCallbackData )
{
    IUnknown    * pData   = (IUnknown*)pCallbackData;
    CActionData * pAction = NULL;

    // Determine if this is in a format we're aware of
    if ( FAILED( pData->QueryInterface( IID_CActionData, (void**)&pAction ) ) )
         return D3D_OK; // Just return unhandled.
```

Now that we have an interface to our CActionData object, we check its m_Action member variable. This can currently only be set to ACTION_NONE or ACTION_PLAYSOUND, but you may decide to add different types of actions to this class yourself. Our callback function only handles sound effect callback keys at the moment, so we have to make sure that we only work with CActionData objects that are in ACTION_PLAYSOUND mode.

```
    // This is our type, lets handle it
    switch ( pAction->m_Action )
    {
        case CActionData::ACTION_PLAYSOUND:

            // Play the sound if available
            if ( pAction->m_pString != NULL )
                 sndPlaySound( pAction->m_pString, SND_ASYNC | SND_NODEFAULT );
            break;

    } // End Switch Action Type

    // Release the action we queried.
    pAction->Release();

    // We're all done
    return D3D_OK;
}
```

If this is an ACTION_PLAYSOUND mode CActionData object, we fetch the filename from its m_pString member and pass it into a Win32 function (sndPlaySound) which loads and plays the specified wave file. We then release the interface to the CActionData object and return from the function.

In our lab project we register two callback keys of type ACTION_PLAYSOUND. Thus, this callback function will be called twice by the controller in a given loop of animation. Since the animation is set to loop, these same two callback keys will cause this function to be called over and over again so that the sounds play during every loop of the animation.

sndPlaySound

To keep things simple, we are using the Win32 function sndPlaySound to play our sound effects. This function takes two parameters. The first is the name of the wave file we wish to play and as you can see, we pass in the name of the wave file stored inside the CActionData object. The second parameter is a combination of one or more of the following flags:

Value	Meaning
SND_ASYNC	The sound is played asynchronously and the function returns immediately after beginning to play the sound. To terminate an asynchronously played sound, call **sndPlaySound** with *lpszSoundName* set to NULL. We use this option so that we can start a long sound playing and continue with our application processing.
SND_LOOP	The sound plays repeatedly until **sndPlaySound** is called again with the *lpszSoundName* parameter set to NULL. You must also specify the SND_ASYNC flag to loop sounds.
SND_MEMORY	If this flag is specified, then the first parameter should not point to a string containing the filename of the wave file but should instead point to an image of a waveform sound in memory.
SND_NODEFAULT	If the sound cannot be found, the function returns silently without playing the default sound. We use this option so that in the event that the sound file is missing, we do not end up playing some default sound instead that might be out of context with our scene.
SND_NOSTOP	If a sound is currently playing, the function immediately returns FALSE, without playing the requested sound. If you specify this flag and a sound is already playing, no action will be taken. Without this flag, the currently playing sound would be cut short and the new sound played immediately.
SND_SYNC	The sound is played synchronously and the function does not return until the sound ends. We do not want to use this flag because our animation would freeze and our application would receive no frame updates until the sound had finished playing.

This is certainly not a function that you would use in a commercial game project (that is what DirectXSound is for), especially since it can only play one sound at a time. However, the purpose of this exercise is to demonstrate the callback mechanism, so this works nicely for us.

Note: In order to use the sndPlaySound function, you must make sure that you are linking to the winmm.lib library (part of the platform SDK). This library includes all the Win32 multimedia functionality.

Advancing our Actor

Let us look at an example of how the actor is updated in our application. You should be fairly comfortable by now with our application framework and the fact that we usually handle any scene based animation updates in the CScene::AnimateObjects method. This method is called for every iteration of our game loop by the CGameApp::FrameAdvance method, giving the scene a chance to update the positions of any objects prior to rendering. The function is passed the application CTimer object so that it can retrieve the elapsed time since the last frame update and forward this time to the CActor::AdvanceTime method.

Note: This is not the exact CScene::AnimateObjects call found in Lab Project 10.1, but it is very close. We have to discuss some additional topics before we are ready to look at the full version of this function. However, we will see all but a few lines of the code and show the important points for this discussion.

```
void CScene::AnimateObjects( CTimer & Timer )
{
    static CActionHandler Handler;

    // Process each object
    for ( ULONG i = 0; i < m_nObjectCount; ++i )
    {
        // Increase actor's reference time if this stores an actor
        if ( m_pObject[i]->m_pActor )
         m_pObject[i]->m_pActor->AdvanceTime( Timer.GetTimeElapsed(),
                                              false,
                                              &Handler );

    } // Next Object
}
```

The function starts by declaring a variable of type CActionHandler. This object contains the callback function that we wish the controller to use to process callback keys. Since we will need to use this object every frame, we make it a static variable rather than a stack variable that has to be repeatedly created and destroyed. We did not make this a member of the CScene class because the only time this object is ever used is in this function. So we decided to keep its scope safely limited to this function. By making it static we can have the best of both worlds -- a global variable that is created once with a safe scope limited to this one function.

We then loop through each CObject in the scene's object array. You will recall from the previous lab project that our scene may consist of single mesh objects and objects with actors attached and still behave properly. In this function however, we are only interested in objects which are actors. So for each object that has an actor attached, we call the actor's AdvanceTime method, passing in the elapsed time retrieved from the timer along with a pointer to our callback handler. At the end of this loop, each actor in the scene will have had its frame hierarchy animated. In Lab Project 10.1 we only have a single actor which will contain all of the objects in the scene and their associated animations.

Why are we passing in false as the second parameter? Recall from our earlier discussion that if it is set to true, then the actor will automatically traverse the hierarchy and build the absolute matrices for each frame. You might imagine that this is surely something we want to have done, right?

While it is true that we need to update the absolute matrices after animation has been applied, we certainly do not wish to perform multiple traversals if we can avoid it. You will recall from the previous lab project that before rendering an actor we must also set its world matrix so that we can position it in the world at an arbitrary location and orientation. (The application can move and position the actor in the scene by applying transformations to the matrix stored in the parent CObject to which the actor is attached.) In the CScene::Render method, prior to rendering the actor's subsets, we call CActor::SetWorldMatrix and pass in the parent CObject matrix as the root node matrix:

```
pActor->SetWorldMatrix( &m_pObject[i]->m_mtxWorld, true );
```

As we learned in the previous workbook, this function is passed the world matrix of the actor as its first parameter and it caches this matrix internally. It is by manipulating the CObject::m_mtxWorld matrix that the application can move the entire actor about in the scene. If true is passed as the second parameter, then the passed matrix is combined with the root node matrix and all changes are filtered down through the hierarchy. Since this call happens in the render function (after the AnimateObjects method), it would be wasteful for us to pass true to the CActor::AdvanceTime method. The result would be a redundant and pointless hierarchy traversal. If we passed true to CActor::AdvanceTime, the hierarchy frames would be generated and the absolute world matrices for each frame constructed and stored. However, when CScene::Render calls CActor::SetWorldMatrix, the absolute matrices we just generated would be obsolete because we have just re-positioned the root node of the actor. Thus, all absolute frame matrices would need to be rebuilt again. Many of the actor methods expose this UpdateFrames Boolean option, but you only want to pass true to the final one called before rendering.

To summarize, using our actor class requires the following steps (some of which are optional):

1. Create a CActor object.
2. Use CActor::RegisterCallback method to register any callback functions for callback key registration and texture and material parsing. (OPTIONAL).
3. Set the desired maximum extents of the actor using the CActor::SetActorLimits method. (OPTIONAL)
4. Load the X file into the actor using CActor::LoadActorFromX.
5. During every iteration of your game loop:
    a. Increment the actor's internal clock using CActor::AdvanceTime (will update the actor's frame matrices).
    b. Set the world matrix for the actor using CActor::SetWorldMatrix.
    c. Render the actor by looping through each subset and calling CActor::DrawSubset.

CActor Utility Functions

We have already discussed the core functions of the actor and saw how easy it is to operate via its interface. However, the actor must also expose many utility functions to allow the application to get and set the properties of the underlying animation controller. For example, the application will want the ability to assign different animation sets to different tracks and to control the global time of the controller. The application will also want to be able to register sequencer events and change the properties of a mixer track to control how animation sets are blended. These are all methods exposed by the ID3DXAnimationController interface which are now (for the most part) wrapped by CActor. Let us take a look at them now.

CActor::ResetTime

CActor::ResetTime can be called to reset the global clock of the animation controller to zero. The reason an application might want to do this is related to the fact that with each call to CActor::AdvanceTime, the global time of the controller gets higher and higher. If the application was left running for a very long time, this global time would eventually reach a very high number. If left long enough, it might loop round to zero by itself, causing jumps in the animation. Furthermore, as sequencer events are registered on the global timeline, this is our only means for returning the global clock of the controller back to zero so that those sequencer events can be triggered again.

This function wraps the ID3DXAnimationController::ResetTime method. The reset time method is a little more complicated than it might first seem. Not only does it reset the global timer of the controller back to zero, but it also wraps around each of the track timers. It does not simply set them to zero since this would cause jumps in the animation. Instead it wraps them around such that the track timer is as close to zero as possible without altering the periodic position of the animation set currently being played. This function is designed to reset the global time to 0.0 while retaining the current periodic position for all playing tracks.

```
void CActor::ResetTime( bool UpdateFrames /* = false */ )
{
    if ( !IsLoaded() ) return;

    // Set the current time if applicable
    if ( m_pAnimController ) m_pAnimController->ResetTime( );

    // Update the frame matrices
    if ( UpdateFrames ) UpdateFrameMatrices( m_pFrameRoot, &m_mtxWorld );
}
```

Notice how this function also has the UpdateFrames Boolean so that the application can force an immediate rebuild of the hierarchy's absolute frame matrices. For example, if this is the last CActor method you are calling prior to rendering the actor's subset you might decide to do this. Normally, you will pass false as this parameter (the default).

CActor::GetTime

This method wraps the ID3DXAnimationController::GetTime method. It returns the current accumulated global time of the animation controller.

```
double CActor::GetTime( ) const
{
    if ( !m_pAnimController) return 0.0f;
    return m_pAnimController->GetTime();
}
```

The GetTime method is useful if an animation wants to perform its own looping logic. For example, imagine that a given animation plays out for 25 seconds and that the track the animation set is assigned to has several sequencer events registered within that time frame. Because sequencer events are on the global timeline, even if the animation set was configured to loop, each sequencer event would only be triggered once. Once the global time of the controller increased past 25 seconds, it would continue to grow and the events would never be triggered again. However, if the application wants these sequencer events to loop with the animation set, it could retrieve the global time of the actor using CActor::GetTime and, if it has climbed over a certain threshold, reset back to zero. This would cause the time to reset every 25 seconds in our example. Thus, the animation set would loop and the sequencer events on the global timeline would be triggered each time.

CActor::GetMaxNumTracks

This function returns the number of tracks currently available on the animation controller. It is a wrapper around the ID3DXAnimationController::GetMaxNumTracks method. For example, the application might use this method to query the current number of tracks available on the mixer and if not enough, the application could call the CActor::SetActorLimits method to extend this numer.

```
ULONG CActor::GetMaxNumTracks( ) const
{
    if ( !m_pAnimController ) return 0;
    return m_pAnimController->GetMaxNumTracks();
}
```

Assigning Animation Sets to Mixer Tracks

The application will often need the ability to assign different animation sets to tracks on the animation mixer. For example, imagine the actor is a game character. When the actor is in a walking state, the application would probably just assign a walking animation set to a mixer track and play that track on its own. But if the character were to suddenly fire its weapon, the application would also want to assign a firing animation set to another track and blend the two animations together so that the character appears to continue walking and fire his weapon. The ID3DXAnimationController interface exposes a single method for this purpose called ID3DXAnimationController::SetTrackAnimationSet. This function

accepts a track number and a pointer to the animation set interface you wish to assign to that track. This must be an animation set that has been previously registered with the animation controller.

Setting a track in this way can sometimes seem a rather clunky approach. Often, our application may not have a pointer to the interface of the animation set, which means it must be retrieved from the controller first (performing a search by name or by index). It would be much nicer if our actor also allowed the application to specify the name of an animation set or its index as well. For example, if we know that we have an animation set called 'Fire' which we wish to assign to track 1, we would first have to retrieve an interface pointer by calling ID3DXAnimationSet::GetAnimationSetByName, and then pass the returned animation set interface into the SetTrackAnimationSet interface. To remove this burden from the application, our CActor actually exposes three methods (SetTrackAnimationSet, SetTrackAnimationSetByName, and SetTrackAnimationSetByIndex) that can be used to assign animation sets to mixer tracks in a variety of different ways.

These methods allow the application to bind an animation set to a mixer track using either its interface, its numerical index within the controller or by using the name of the animation set. They are shown below with a brief discussion of each.

CActor::SetTrackAnimationSet

This method is a wrapper around its ID3DXAnimationController counterpart. The application must pass two parameters; the first should be the zero based index of the mixer track that the specified animation set should be assigned to and the second should be the ID3DXAnimationSet interface pointer we wish to set.

```
HRESULT CActor::SetTrackAnimationSet(ULONG TrackIndex,
                                     LPD3DXANIMATIONSET pAnimSet )
{
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set anim set
    return m_pAnimController->SetTrackAnimationSet( TrackIndex, pAnimSet );
}
```

This function forwards the parameters to the ID3DXAnimationController::SetTrackAnimationSet method. In order to use this method, the application must already have an interface to the animation set it wishes to assign to a track. We can use the CActor::GetAnimationSet method (discussed in a moment) for this purpose.

CActor::SetTrackAnimationSetByIndex

We implemented this function to make the assignment of animation sets to mixer tracks a one line call within the application code. If the application knows the zero based index of the animation set it wishes to assign, then this function is the one to use. The application passes the track to assign the animation set to and the index of the animation set.

```
HRESULT CActor::SetTrackAnimationSetByIndex( ULONG TrackIndex, ULONG SetIndex )
{
    HRESULT hRet;
    LPD3DXANIMATIONSET pAnimSet = NULL;

    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Retrieve animation set by index
    hRet = m_pAnimController->GetAnimationSet( SetIndex, &pAnimSet );
    if ( FAILED(hRet) ) return hRet;

    // Set anim set
    hRet = m_pAnimController->SetTrackAnimationSet( TrackIndex, pAnimSet );

    // Release the one we retrieved
    pAnimSet->Release();

    // Return result
    return hRet;
}
```

This method automates some of the work the application would usually have to do. It calls the ID3DXAnimationController::GetAnimationSet method to retrieve an interface to the animation set at the specified index. As discussed in the textbook, this method accepts an index and the address of an ID3DXAnimationSet interface pointer. If a valid animation set exists in the controller at the specified index, its interface will be returned in the second parameter. We then pass this interface into the ID3DXAnimationController::SetTrackAnimationSet method, along with the track index, to perform the actual track assignment. Notice that we release the interface that was returned from ID3DXAnimationController::GetAnimationSet so the object's reference count is correctly maintained.

CActor::SetTrackAnimationSetByName

This final track setting method is almost identical to the previous function in way that it behaves. This time, the function should be passed a track index and a string containing the name of the animation set we wish to assign to the specified mixer track.

The function uses the ID3DXAnimationController::GetAnimationSetByName method, which is passed the name of the animation set and the address of an ID3DXAnimationSet interface pointer. If an animation set with the specified name exists within the animation controller, on function return, the pointer passed as the second parameter will point to the interface of this animation set. Once the animation set interface has been retrieved, we can bind it to the desired track using the ID3DXAnimationController::SetTrackAnimationSet method as we have done in the previous two methods.

```
HRESULT CActor::SetTrackAnimationSetByName( ULONG TrackIndex, LPCTSTR SetName )
{
    HRESULT hRet;
    LPD3DXANIMATIONSET pAnimSet = NULL;
```

```
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Retrieve animation set by name
    hRet = m_pAnimController->GetAnimationSetByName( SetName, &pAnimSet );
    if ( FAILED(hRet) ) return hRet;

    // Set anim set
    hRet = m_pAnimController->SetTrackAnimationSet( TrackIndex, pAnimSet );

    // Release the one we retrieved
    pAnimSet->Release();

    // Return result
    return hRet;
}
```

Using CActor to Alter Mixer Track Properties

As discussed in the textbook, the animation controller will have an internal multi-track animation mixer available for use. Much like an audio mixing desk, each track has a number of properties that can be altered to change the way the animation set is played and combined into the final result. We can enable or disable a track midway through an animation, alter the track position to adjust the timeline of a single animation set, change the speed at which the track position is incremented with respect to the global timer, and change the weight of a track so that the SRT data contributes more or less to the final matrix generated for that frame.

The ID3DXAnimationController provides methods to allow our application to perform all of these tasks, and so too will our CActor. In the next section, we will look at a series of small wrapper functions around the corresponding animation controller methods that manage the animation mixer properties and behavior.

CActor::SetTrackPosition

This method allows the application to alter the timer of a single track on the mixer. We pass in the index of the track we wish to alter and the new position we would like to set the track's timer to. Remember that this is track time (not periodic animation set time) that we are setting here. Since the track position is mapped to a periodic position, altering the track position allows us to manually jump, skip, or pause an animation. For example, if we called this function in our game loop passing in the same track position each time, the animation would appear frozen. This is because the same track position would be mapped to the same periodic position resulting in identical SRT data. Of course, if the animation set was also being blended with other tracks, only its contribution would be frozen. The other tracks would continue to play out as expected.

```
HRESULT CActor::SetTrackPosition( ULONG TrackIndex, DOUBLE Position )
{
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;
```

```
    // Set the position within the track.
    return m_pAnimController->SetTrackPosition( TrackIndex, Position );
}
```

CActor::SetTrackEnable

This function is passed a track index and a Boolean variable. If the Boolean is set to true, the track will be enabled and any animation set assigned to that track will play out as expected. The track will remain enabled until it is disabled. If false is passed, the specified track will be disabled and will remain so until it is re-enabled by the application. When a track is disabled, it will be ignored by the animation controller. Any animation set assigned to that track will not contribute to the frame hierarchy the next time AdvanceTime is called.

```
HRESULT CActor::SetTrackEnable( ULONG TrackIndex, BOOL Enable )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set the details
    return m_pAnimController->SetTrackEnable( TrackIndex, Enable );
}
```

CActor::SetTrackPriority

In the textbook we learned that each track on the mixer can be assigned to one of two priority blend groups. The fact that these groups are referred to as high and low priority is perhaps a bit misleading as the blend ratio between these two groups can be altered such that the low priority group could contribute more to the frame hierarchy than the high priority group.

This method allows us to assign a track to one of the two priority groups. We specify the priority group using the D3DXPRIORITY_TYPE enumeration shown below:

```
typedef enum _D3DXPRIORITY_TYPE
{
    D3DXPRIORITY_LOW = 0,
    D3DXPRIORITY_HIGH = 1,
    D3DXEDT_FORCE_DWORD = 0x7fffffff
} D3DXPRIORITY_TYPE;
```

```
HRESULT CActor::SetTrackPriority( ULONG TrackIndex, D3DXPRIORITY_TYPE Priority )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set the details
    return m_pAnimController->SetTrackPriority( TrackIndex, Priority );
}
```

CActor::SetTrackSpeed

We discussed in the textbook that each track has its own independant position and its own speed (set to 1.0 by default). Using this method we can speed up/slow down the rate at which a single animation set is playing without affecting animation sets assigned to any other tracks on the mixer. The application passes this function the index of the track and the new speed.

```
HRESULT CActor::SetTrackSpeed( ULONG TrackIndex, float Speed )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set the details
    return m_pAnimController->SetTrackSpeed( TrackIndex, Speed );
}
```

The track speed is changed internally by the controller. It uses the track speed property to scale the elapsed time passed into the AdvanceTime method before adding the result to the track position. As a simple example, let us imagine we pass in an elapsed time of 5 seconds. Five seconds would be added to the global clock of the animation controller. Now the controller needs to add this elapsed time to the position of each track to update the track timelines also. However, before it does this, it scales elapsed time by the track speed. If we imagine that the track speed was set to 3.0 and that the current track position was 20.0, the track position would be updated like so:

TrackPosition =20
ElapsedTime = 5
Track Speed = 3.0

NewTrackPosition = TrackPosition + (ElapsedTime * TrackSpeed)
= 20 + (5 * 3)
= 20 + 15
= 35

If we imagine that another currently active mixer track also has a track position of 20 but only has a track speed of 0.5 we can see that the track position for this track would be updated much less during thr same advance time call:

TrackPosition = 20
ElapsedTime = 5
Track Speed = 0.5

NewTrackPosition = TrackPosition + (ElapsedTime * TrackSpeed)
= 20 + (5 * 0.5)
= 20 + 2.5
= 22.5

As you can see, the first animation set will appear to play much quicker than the second one in a single update. In the first case, with an elapsed time of 5 seconds, the timeline of the track was actually updated 15 seconds to a new track position of 35 seconds. The second animation set was advanced only by 2.5 seconds to a new track position of 22.5 seconds.

CActor::SetTrackWeight

The weight of a track controls how strongly the SRT data generated by its assigned animation set contributes to the overall hierarchy. It also controls the weight at which it is blended with other animation sets/tracks assigned to the same priority group.

```
HRESULT CActor::SetTrackWeight( ULONG TrackIndex, float Weight )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set the details
    return m_pAnimController->SetTrackWeight( TrackIndex, Weight );
}
```

The default weight for each track is 1.0. This means that the SRT data generated by the animations of the animation set will not be scaled and will influence the hierarchy at full strength. If the weight of track A is set to 0.5 and the weight assign to track B is 1.0 and they both belong to the same priority group, the animation set assigned to track B will influence the hierarchy more strongly than track A. Setting track weights is very important as it allows you to find the right balance when multiple animations are being blended and played. We will see many examples of multi-track animation blending as we progress through the course.

CActor::SetTrackDesc

The actor also exposes a function that allows the application to set all the track properties with a single function call. This function is a wrapper around ID3DXAnimationController::SetTrackDesc. It accepts a track index and a pointer to a D3DXTRACK_DESC structure which contains the new properties for the track

```
HRESULT CActor::SetTrackDesc( ULONG TrackIndex, D3DXTRACK_DESC * pDesc )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set the details
    return m_pAnimController->SetTrackDesc( TrackIndex, pDesc );
}
```

Here we see the D3DXTRACK_DESC structure as a reminder:

```
typedef struct _D3DXTRACK_DESC {
    D3DXPRIORITY_TYPE Priority;
    FLOAT Weight;
    FLOAT Speed;
    DOUBLE Position;
    BOOL Enable;
} D3DXTRACK_DESC, *LPD3DXTRACK_DESC;
```

Retrieving the Properties of a Mixer Track

Just as an application using our actor will need the ability to set track properties on the animation mixer, it will also need a means for retrieving the current values being used by those mixer tracks. The ID3DXAnimationController exposes two methods that allow the application to retrieve the properties of the mixer.

CActor::GetTrackDesc

CActor::GetTrackDesc is the exact opposite of the method previously discussed. The application passes in the track number it wishes to retrieve properties for and the address of a D3DXTRACK_DESC structure. On function return, this structure will be filled with the properties of the specified mixer track.

```
HRESULT CActor::GetTrackDesc( ULONG TrackIndex, D3DXTRACK_DESC * pDesc ) const
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Get the details
    return m_pAnimController->GetTrackDesc( TrackIndex, pDesc );
}
```

CActor::GetTrackAnimationSet

The second method exposed by the animation controller to communicate track properties to the application is ID3DXAnimationController::GetTrackAnimationSet. This allows an application to retrieve an interface to the animation set currently assigned to a mixer track.

This function is passed only a track index and returns a pointer to an ID3DXAnimationSet interface for that animation set assigned to the track.

```
LPD3DXANIMATIONSET CActor::GetTrackAnimationSet( ULONG TrackIndex ) const
{
    LPD3DXANIMATIONSET pAnimSet;

    // Validate
    if ( !m_pAnimController ) return NULL;

    // Get the track details (calls AddRef) and return it
    if ( FAILED(m_pAnimController->GetTrackAnimationSet( TrackIndex, &pAnimSet )))
        return NULL;
    return pAnimSet;
}
```

Setting the Priorty Blend Weight of the Controller

The animation controller has a global property for all tracks called the priority blend weight. It is a single scalar value (usually between 0.0 and 1.0) which determines how the high and low priority track groups on the mixer will be combined during the final blending stage. Setting this value to 0.5 for example, will blend the SRT data generated from both blend groups using a 50/50 mix.

Our actor wraps the ID3DXAnimationController methods for setting and retrieving this blend weight. CActor::SetPriorityBlend and CActor::GetPriortyBlend are two methods exposed to the application for this purpose:

```
HRESULT CActor::SetPriorityBlend( float BlendWeight )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Set the details
    return m_pAnimController->SetPriorityBlend( BlendWeight );

}
```

```
float CActor::GetPriorityBlend( ) const
{
    // Validate
    if ( !m_pAnimController ) return 0.0f;

    // Get the details
    return m_pAnimController->GetPriorityBlend( );
}
```

Querying the Controller's Animation Sets

The ID3DXAnimationController interface exposes methods to allow the application to fetch interfaces for registered animation sets. Unlike the ID3DXAnimationController::GetTrackAnimationSet method, we can use these methods to fetch interfaces for animation sets that are not currently assigned to any track on the animation mixer. Obviously we need to be able to fetch animation sets which are not currently assigned to mixer tracks so that we can assign them to tracks on the mixer. The controller also exposes methods that allow the application to query the number of animation sets currently registered with the controller as well as the maximum number supported.

This collection of functionality is covered by four different actor methods. The first method allows the application to query the maximum number of animation sets that can be registered with the animation controller simultaneously:

```
ULONG CActor::GetMaxNumAnimationSets( ) const
{
    if ( !m_pAnimController ) return 0;
    return m_pAnimController->GetMaxNumAnimationSets( );
}
```

The second method allows the application to query the number of animation sets that are currently registered with the animation controller:

```
ULONG CActor::GetAnimationSetCount( ) const
{
    // Validate
    if ( !m_pAnimController ) return 0;

    // Return the count
    return m_pAnimController->GetNumAnimationSets( );
}
```

The next two methods allow us to fetch interfaces to animation sets that are currently registered with the controller. The first method allows us to fetch the animation set interface using an index. This is the zero based index of the animation set within the controllers's array of registered animation sets:

```
LPD3DXANIMATIONSET CActor::GetAnimationSet( ULONG Index ) const
{
    LPD3DXANIMATIONSET pAnimSet;

    // Validate
    if ( !m_pAnimController ) return 0;

    // Get the animation set and return it
    if ( FAILED( m_pAnimController->GetAnimationSet( Index, &pAnimSet ) ))
        return NULL;
    return pAnimSet;
}
```

Your application may not always know the index of an animation set it wishes to fetch, but may know the name of the animation. This second function allows us to pass in a string containing the name of the animation set we wish to retrieve an interface for. As with the above function, this one wraps the call to its ID3DXAnimationController counterpart. If an animation set is registered with the animation controller which matches the name passed into the function, an ID3DXAnimationSet interface to that animation set will be returned:

```
LPD3DXANIMATIONSET CActor::GetAnimationSetByName( LPCTSTR strName ) const
{
    LPD3DXANIMATIONSET pAnimSet;

    // Validate
    if ( !m_pAnimController ) return 0;

    // Get the animation set and return it
    if ( FAILED( m_pAnimController->GetAnimationSetByName( strName, &pAnimSet ) ))
```

```
        return NULL;
    return pAnimSet;
}
```

Registering Track Events with the Animation Sequencer

As discussed in the textbook, the animation controller includes an event sequencer that allows an application to schedule track property changes for a specific global time. Just as each property of a mixer track can be set with a call to one of the SetTrackXX methods discussed ealier, each track property also has a method in the ID3DXAnimationController for scheduling that track property change to occur at a specific time. Our CActor has wrapped these methods and the source code to these functions will be dicussed in this section. Since these are simply wrapper functions around the methods of the animation controller (discussed in the main course text), these functions will requite little explanation.

> **Note:** When using these sequencer functions, remember that events are specified in global time. That is, the global time of the controller is incremented with each AdvanceTime call. Global time does not loop unless the timer is specifically reset using CActor::ResetTime. Also remember that the global time of the controller may be very different from the time of a specific track. This will be true if the track position has been manipulated by the application (see CActor::SetTrackPosition) or if the speed of a track is not the default. If the global time is never reset and continues to increment, a registered sequencer event will only ever occur once. This is true even if the animation set assigned to that track is configured to loop.

CActor::KeyTrackEnable

This method can be used by the application to schedule a track to be enabled or disabled at a specific global time. The application should pass in the track index as the first parameter and the second parameter should be set to true or false indicating whether the track should be enabled or disabled by this event. The third parameter is the global time at which this event should occur.

```
D3DXEVENTHANDLE CActor::KeyTrackEnable( ULONG TrackIndex, BOOL NewEnable,
                                        double StartTime )
{
    // Validate
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->KeyTrackEnable( TrackIndex, NewEnable, StartTime );
}
```

Our method simply forwards the request on to the animation controller. This method is a sequencer version of the CActor::SetTrackEnable method discussed previously.

CActor::KeyTrackPosition

This method allows the application to schedule a track position manipulation at a specified global time. Essentially it allow us to manipulate track time at a specified global time. It is important that we distinguish between the two different timelines being used here. As the first paramater the track index is passed for which the position will be changed. As the second parameter we pass in the new track position that we would like this track to be set to at the time when this event is triggered.

The 'NewPosition' is specified in track time, not global time and not as a periodic position. We might for example set this NewPosition to 20 seconds for example. If the animation had a period of 10 seconds, then this would essentially cause the animation to jump to the beginning of its 3rd iteration (if looping is enabled for this animation set). This is because a track position of 20 seconds would be mapped to a periodic position of 0 seconds for a looping animation with a 10 second period.

The third parameter is the time at which we would like this track position change event to occur. Unlike the second parameter, this is not specified in track time but is instead specified in global time which might be quite different. As mentioned, the sequencer events are registered on the global time line.

```
D3DXEVENTHANDLE CActor::KeyTrackPosition( ULONG TrackIndex, double NewPosition,
                                          double StartTime )
{
    // Validate
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->KeyTrackPosition(TrackIndex,NewPosition,StartTime);
}
```

This method is the sequencer version of the CActor::SetTrackPosition method discussed previously.

CActor::KeyTrackSpeed

This method allows the application to schedule a track speed property change to occur at a specified global time. We pass in the track index and the new speed we would like this track to be set to when the event is triggered. The third parameter is the global time at which we would like this event to be triggered. The fourth parameter is the event duration (in seconds). The duration describes how many seconds we would like it to take, after the event has been triggered, to transition the current speed to the new speed. This feature (along with the next parameter) allows us to schedule smoother transitions.

```
D3DXEVENTHANDLE CActor::KeyTrackSpeed( ULONG TrackIndex, float NewSpeed,
                                       double StartTime, double Duration,
                                       D3DXTRANSITION_TYPE Transition )
{
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->KeyTrackSpeed( TrackIndex, NewSpeed,
                                             StartTime, Duration, Transition );
}
```

The fifth parameter is a member of the D3DXTRANSITION_TYPE enumeration which describes how we would like the change from the current speed to the new speed to occur over the specified duration. The two choices are linear interpolation and spline based interpolation.

```
typedef enum _D3DXTRANSITION_TYPE {
    D3DXTRANSITION_LINEAR = 0x000,
    D3DXTRANSITION_EASEINEASEOUT = 0x001,
    D3DXEDT_FORCE_DWORD = 0x7fffffff
} D3DXTRANSITION_TYPE;
```

Whichever transition type we use, the change from the old speed to the new speed will still be performed within the specified duration. If linear is being used then the current speed will linearly ramp up/down from the old speed to the new speed over the duration. If ease-in ease-out is used, the speed change will start off slowly, gradually increase until some maximum, level off, and then, when near the end of the specified duration, the speed change will decrease until the final target is reached. As the member of the enumerated type suggests, this changes the speed in a more natural way, speeding up and slowing down at the outer ends of the duration.

This method is the sequencer version of the SetTrackSpeed method discussed previously.

CActor::KeyTrackWeight

This method allows the application to schedule a weight change for a specified track at a specified global time. By sequencing weight change events, we can essentially script the strength at which an animation set influences the frame hierarchy over a period of time.

```
D3DXEVENTHANDLE CActor::KeyTrackWeight( ULONG TrackIndex, float NewWeight,
                                        double StartTime, double Duration,
                                        D3DXTRANSITION_TYPE Transition )
{
    // Validate
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->KeyTrackWeight( TrackIndex, NewWeight, StartTime,
                                              Duration, Transition );
}
```

This function, like the previous one, also accepts a duration and transition type parameter. This allows the weight of the track to be changed from its current weight (at the time the event is triggered) to its new weight, over a specified time in seconds (the Duration parameter). The transition type once again controls whether we wish the change from the current weight to the new weight of the track to be interpolated linearly or using a spline based approach.

This method is the sequencer version of the CActor::SetTrackWeight method discussed previously.

Registering Priorty Blend Changes with the Sequencer

The animation controller has a global property called the *priority blend*. This value is usually set between zero and one and is used to weight the contributions of the high and low priority groups that are blended together in the final mixing stage.

As discussed in the textbook, mixer tracks can be assigned to one of two priority groups (high or low). When AdvanceTime is called, the SRT data for each animation set is generated. Once we have the SRT data generated from each track, all SRT data belong to tracks in the same priority groups are blended together using their track weights to weight their contribution in the mix. This is done for both the high and low priority groups. At this point, the animation controller has two groups of SRT data: the output from the low priority mixing phase and the output from the high priority mixing phase. Once this is output is prepared, the SRT data in these two groups is blended together. The priority blend value of the controller is used to weight the contribution of the two groups in the final mix. The output of this final stage is a single set of SRT data used to rebuild the relative matrix of the attached animated frame.

```
D3DXEVENTHANDLE CActor::KeyPriorityBlend( float NewBlendWeight, double StartTime,
                                          double Duration,
                                            D3DXTRANSITION_TYPE Transition )
{
    // Validate
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->KeyPriorityBlend( NewBlendWeight,
                                                StartTime, Duration, Transition );
}
```

Just like the track speed and weight changes, when we schedule an event to change the priority blend weight, we can also specify a duration and transition type. This allows us to slowly change the priority blend weight from its old value to the new one.

This method is the sequencer version of the CActor::SetPriorityBlend method discussed previously.

Removing Sequencer Events from the Global Timeline

There may be times when you need to un-register a sequencer event. Perhaps your application has reset the global time of the controller but does not wish a certain event to be triggered again the second time through. This next series of methods allow the application to unregister sequencer events in a variety of different ways.

CActor::UnkeyEvent

The ID3DXAnimationController has an UnkeyEvent method that we have wrapped in CActor::UnkeyEvent. The method is passed the event handle of the event you wish to unregister. All

event registration functions return a D3DXEVENTHANDLE, which is a unique ID for that sequencer event. If you wish to perform some action on that event later (e.g., remove it), this is the handle that you can use to identify this specific event to the D3DX animation system.

If the passed D3DXEVENTHANDLE matches the handle of an event currently registered on the global timeline, this method will remove it. This method can be used to unregister all event types (weight, speed, position, etc). The animation controller will automatically know which type of event it is and which track that event has to be removed from based on the passed event handle.

```
HRESULT CActor::UnkeyEvent( D3DXEVENTHANDLE hEvent )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Send the details
    return m_pAnimController->UnkeyEvent( hEvent );
}
```

CActor::UnkeyAllTrackEvents

This method removes all events that currently exist for a given track. The application simply passes in the index of the track and the controller will remove any events that have been registered for that track.

```
HRESULT CActor::UnkeyAllTrackEvents( ULONG TrackIndex )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Send the details
    return m_pAnimController->UnkeyAllTrackEvents( TrackIndex );
}
```

Events that have been registered for other mixer tracks are not affected by this method. Note as well that priority blend events are *not* removed by this method. This is because priority blend events are not specific to a given track but instead alter the priority blend weight of the animation controller, which is used by all tracks. Any tracks events that were previously registered with the track index passed into the function will no longer be valid when the function returns.

CActor::UnkeyAllPriorityBlends

Priority blend events are unlike all other event types because they are not track specific. Instead, they alter a global property of the controller. Because of this, the CActor::UnkeyAllTrackEvents method cannot be used to unregister events of this type. The CActor::UnkeyAllPriorityBlends method should be used to clear all priority blend events from the global timeline.

```
HRESULT CActor::UnkeyAllPriorityBlends( )
{
    // Validate
```

```
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Send the details
    return m_pAnimController->UnkeyAllPriorityBlends( );
}
```

Validating Events

The CActor::ValidateEvent function can be used to test if an event is still valid. Valid events are events which have not yet been executed (because the global time is smaller than the timestamp of the event) or events that are currently being executed. The method accepts a single parameter, the D3DXEVENTHANDLE of the event you wish to validate. Any event whose timestamp is less than the global time of the controller is considered invalid.

```
HRESULT CActor::ValidateEvent( D3DXEVENTHANDLE hEvent )
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Send the details
    return m_pAnimController->ValidateEvent( hEvent );
}
```

When the global time of the controller is reset, all events that were previously considered invalid become valid again. The exception of course, is if you pass in the handle of an event that your application has since un-registered.

Fetching the Details for an Upcoming Event

It is often useful for an application to know when a sequencer event is currently being executed so that the application can perform some action when the event is triggered. This is much like how an application might respond to a callback function. However, a callback method is not used for this purpose. Instead, the application can poll for an event type on a specific track or poll for a currently executing priority blend event using this next series of functions. If an event is currently being executed that fits the specified criteria, the handle of the event will be returned. The CActor::GetEventDesc method can then be used to fetch the event details of this event using the passed event handle.

CActor::GetCurrentTrackEvent

This method is used to test if an event is currently being executed on a given track. The function accepts two parameters. For the first parameter, the application should pass the index of the track for which it is requesting event information. For the second parameter, the application should pass a member of the D3DXEVENT_TYPE enumeration specifying the type of events it is interested in being informed about.

This method just wraps the corresponding ID3DXAnimationController method. If an event of the specified type is currently being executed on the specified track, the handle of the event will be returned from the method.

```
D3DXEVENTHANDLE CActor::GetCurrentTrackEvent( ULONG TrackIndex,
                                              D3DXEVENT_TYPE EventType ) const
{
    // Validate
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->GetCurrentTrackEvent( TrackIndex, EventType );
}
```

The D3DXEVENT_TYPE enumeration is defined as follows in the DirectX header files.

```
typedef enum _D3DXEVENT_TYPE {
    D3DXEVENT_TRACKSPEED = 0,
    D3DXEVENT_TRACKWEIGHT = 1,
    D3DXEVENT_TRACKPOSITION = 2,
    D3DXEVENT_TRACKENABLE = 3,
    D3DXEVENT_PRIORITYBLEND = 4,
    D3DXEDT_FORCE_DWORD = 0x7fffffff
} D3DXEVENT_TYPE;
```

Passing D3DXEVENT_PRIORITYBLEND into this method will never return a valid handle since priority blend events are not track events.

CActor::GetCurrentPriorityBlend

This method accepts no parameters and will return the handle of the currently executing priority blend event. If no priority blend event is being executed then the function will return NULL.

```
D3DXEVENTHANDLE CActor::GetCurrentPriorityBlend( ) const
{
    // Validate
    if ( !m_pAnimController ) return NULL;

    // Send the details
    return m_pAnimController->GetCurrentPriorityBlend( );
}
```

CActor::GetEventDesc

This method can be used to fetch the details of any sequencer event registered on the global timeline. It accepts two parameters. The first parameter is the handle of the event for which information is to be retrieved (via parameter two). This could be the handle returned from either the CActor::GetCurrentPriorityBlend or CActor::GetCurrentTrackEvent if you wish to retrieve information

about an event that is currently being executed. The method simply wraps its ID3DXAnimationController counterpart.

```
HRESULT CActor::GetEventDesc( D3DXEVENTHANDLE hEvent, LPD3DXEVENT_DESC pDesc )
const
{
    // Validate
    if ( !m_pAnimController ) return D3DERR_INVALIDCALL;

    // Send the details
    return m_pAnimController->GetEventDesc( hEvent, pDesc );
}
```

CActor Utility Functions

The final two methods of the CActor interface fall into the category of utility functions. These are methods that you are not likely to use very often when integrating CActor into your day-to-day projects, but they do come in handy when using the CActor class as part of a tool. The first method is simply an upgrade from Lab Project 9.1.

CActor::SaveActorToX

This method is used to save the actor out to an X file. Unlike the previous version of this function, we now pass the D3DXSaveMeshHierarchyToFile function the address of the animation controller interface. This will ensure that D3DX will save the frame and mesh data to the X file as well as the animation data. This single change to this function is highlighted in bold in the listing below.

```
HRESULT CActor::SaveActorToX( LPCTSTR FileName, ULONG Format )
{
    HRESULT hRet;

    // Validate parameters
    if ( !FileName ) return D3DERR_INVALIDCALL;

    // If we are NOT managing our own attributes, fail
    if ( GetCallback( CActor::CALLBACK_ATTRIBUTEID ).pFunction != NULL )
         return D3DERR_INVALIDCALL;

    // Save the hierarchy back out to file
    hRet = D3DXSaveMeshHierarchyToFile( FileName, Format,
                                        m_pFrameRoot, m_pAnimController,
                                        NULL );
    if ( FAILED(hRet) ) return hRet;

    // Success!!
    return D3D_OK;
}
```

CActor::ApplySplitDefinitions

This next function is a large one, but the code does not use any techniques we have not yet encountered. Before examining the code, let us first discuss the service this method is intended to provide and why an application would be interested in such a service.

As discussed in the textbook, some X file exporters only support saving animated X files with a single animation set. The Deep Exploration™ plugin for 3D Studio Max™ supports multiple animation set export and is a really excellent X file exporter. If you can afford both 3D Studio and this plugin, you would be hard pressed to find a better and easier solution. However, most students just getting started in their game programming studies simply cannot afford such professional tools. So a workaround for this problem is often needed.

One way to get around this problem requires artist participation. The artist can store all the animation sequences for a given object directly after each other on a single timeline. For example, if an artist was building animation data for a game character, he might use the first 10 seconds of the timeline to store the walking animation, the next 10 seconds to store the idle animation, the next 10 seconds for the death animation, and so on. If this character's animation was played back in this arrangement, it would essentially cycle through all possible animations that the character is capable of as the timeline is traversed from start to finish.

In Lab Project 10.2 we use an X file called 'Boney.x' that has been constructed in this way. Because all the animation sequences are stored on the same timeline and are essentially part of the same animation set (as far as the modeling application is concerned), this X file can be exported by applications that support saving only a single animation set. All the animation data we need will be saved in the X file.

The problem is that all of these sequences will exist in a single animation set and this is far from ideal. Normally, we would like these animations to be stored in separate animation sets so that each individual sequence can be played back by the animation controller in isolation. We would also like the ability to blend, for example, a walking animation sequence with a firing animation sequence. We know that in order to accomplish this, we must have those two sequences in different animation sets so that they can be assigned to different tracks on the animation mixer. But having been supplied with this X file which contains all the animation data in a single set, what are we to do?

Our answer was to implement a new method for CActor called ApplySplitDefinitions, which allows us to correct this problem after the X file has been loaded. This method will allow the application to pass in an array of split definition structures. Each structure will describe how we would like a section of the current 'single' animation set to be copied into a new animation set. For example, if we had a single animation set with five sequences stored in it, this function could be used to break it into five separate animation sets. Each split definition structure that we pass in would contain the start and end positions on the original 'combined' timeline where the new animation set will start and end. It would also contain the name that we would like to call this new animation set. Once the method returns, the original animation set would have been released and we would now have five separate animation sets in its place.

This approach does require that the artist let us know the start and end positions of each sequence in the set so that our code can correctly fill in the split definition structures before passing them to the ApplySplitDefinitions function. However, this can also prove to be a limitation, especially if the artist is not available for us to question. So we have designed something that negates the need for even that requirement.

In Lab Project 10.2 we will develop the source code for a tool called The Animation Splitter. It is a simple dialog box driven GUI tool wrapped around the CActor::ApplySplitDefinitions method. The Splitter will allow us to graphically set markers on the combined timeline and apply the splits into separate sets. Behind the scenes this tool will call the CActor::ApplySplitDefinitions method to split the animation set into multiple animation sets based on the markers you set via the user interface. Then you can save the actor back out to an X file with its newly separated animation sets.

Whether you use this tool to split up your combined animations as a pre-process or use the CActor::ApplySplitDefinitions method in your main code after loading the X file is up to you. Bear in mind that after you have applied the splits, you can always use the CActor::SaveActorToX method to save the data back out to file in its new form. This GUI tool is just a user-friendly way to configure the call to CActor::ApplySplitDefinitions. So let us have a look at this final CActor method which will be used by the animation splitter in the next project to do all the heavy lifting.
The CActor::ApplySplitDefinitions method accepts an array of AnimSplitDefinition structures This structure (defined in CActor.h) is shown below.

```
typedef struct _AnimSplitDefinition // Split definition structure
{
    TCHAR   SetName[128];       // The name of the new animation set
    ULONG   SourceSet;          // The set from which we're sourcing the data
    double  StartTicks;         // Ticks at which this new set starts
    double  EndTicks;           // Ticks at which this new set ends
    LPVOID  Reserved;           // Reserved for use internally, do not use.

} AnimSplitDefinition;
```

Each stucture of this type that we pass into the function will describe a new animation set that we wish to create from a segment of an existing one. The members are described below:

TCHAR SetName[128]
This member is where supply the name of the new animation set we wish to be created.

ULONG SourceSet
In this member we supply the index of the animation set currently registered with the controller that we would like the data for this new animation set to be extracted from. In the case of a single combined animation set, this will always be set to zero since the controller will only have one set. However, the function will handle splitting from multiple sources. This is essentially the source information for this definition. It is the set which will have a section of its keyframe data copied into a new animation set. This set will be released after the splits have been applied.

double StartTicks
This is the starting marker on the source set's animation timeline indicating where the section of animation data we wish to copy into a new set begins. For example, if we were supplied a single animation set and informed that the walking sequence for this animation began at tick 400 and ended at tick 800, we would set both the StartTicks and EndTicks members of this structure to 400 and 800 respectively. Any animation data in the source animation set between these two tick positions will be copied into the new set.

double EndTicks
This member describes the end marker for the section of animation data we wish to copy into the new animation set from the source animation set.

LPVOID Reserved
This member is reserved and should not be used.

Let us first look at an example of how an application might use the CActor::ApplySplitDefinitions function, before we cover the code. Continuing our previously discussed example, let us imagine we have loaded an X file into our actor which contains two animation sequences that we would like to divide into two separate animations sets called 'Walk' and 'Run' respectively. Let us also imagine that we are informed by our artist that the walking sequence is contained in the animation timeline between ticks 0 and 400 and the running sequence is contained in the original animation set between ticks 400 and 800. To perform this action, we would write the following code:

```
// Allocate an array for two definitions
AnimSplitDefinition    AnimDefs[2];
UINT                   NumberOfAnimDefs=2;

// Set up definition 1 called 'Walk'
_tcscpy( AnimDefs[0].SetName, _T("Walk") );
AnimDefs[0].SourceSet   =     0
AnimDefs[0].StartTicks  =     0
AnimDefs[0].EndTicks    =     400

// Set up defnition 2 called 'Run'
_tcscpy( AnimDefs[1].SetName, _T("Run") );
AnimDefs[0].SourceSet   =     0
AnimDefs[0].StartTicks  =     400
AnimDefs[0].EndTicks    =     800

// Create the two new animation sets and release the original ( set 0 )
pActor->ApplySplitDefinitions( AnimDefsCount , AnimDefs );
```

That is all there is to the process. The animation controller of the actor now has two separate animation sets called 'Walk' and 'Run' where previously only a single animation set (set 0) exisited. The original animation set will have been unregistered from the controller and freed from memory when the CActor::ApplySplitDefinitions function returns.

Let us now study the code. This is going to be quite a large function so we will need to cover it a step at a time. Fortunately, there is hardly anything here that we have not studied and implemented already, so for the most part it will be easy going.

The first section of the function has the job of cloning the actor's animation controller if it cannot support the number of animation sets passed in the nDefCount parameter. Remember, each split definition in this array is describing a new animation set we would like to create and register with the animation controller. If the animation controller is currently limited to only having two animation sets registered with it, but the application has passed in an array of 10 split definitions, we know we must have an animation controller that is capable of having the 10 new animation sets we are about to create registered with it.

```
HRESULT CActor::ApplySplitDefinitions( ULONG nDefCount,
                                       AnimSplitDefinition pDefList[] )
{
    HRESULT                     hRet;
    LPCTSTR                     strName;
    ULONG                       i, j, k;
    LPD3DXANIMATIONSET          pAnimSet;
    LPD3DXKEYFRAMEDANIMATIONSET pSourceAnimSet, pDestAnimSet;

    D3DXKEY_VECTOR3            *ScaleKeys     = NULL, *NewScaleKeys = NULL;
    D3DXKEY_VECTOR3            *TranslateKeys = NULL, *NewTranslateKeys = NULL;
    D3DXKEY_QUATERNION         *RotateKeys    = NULL, *NewRotateKeys = NULL;

    ULONG                       ScaleKeyCount, RotationKeyCount,
                                TranslationKeyCount, AnimationCount;

    ULONG                       NewScaleKeyCount,
                                NewRotationKeyCount,
                                NewTranslationKeyCount;

    D3DXVECTOR3                 StartScale, EndScale,
                                StartTranslate, EndTranslate;

    D3DXQUATERNION              StartRotate, EndRotate;

    // Validate pre-requisites
    if ( !m_pAnimController || !IsLoaded() || nDefCount == 0 )
        return D3DERR_INVALIDCALL;

    // Clone our animation controller if there are not enough set slots available
    if ( m_pAnimController->GetMaxNumAnimationSets() < nDefCount )
    {
        LPD3DXANIMATIONCONTROLLER pNewController = NULL;

        // Clone the animation controller
        m_pAnimController->CloneAnimationController(
                                  m_pAnimController->GetMaxNumAnimationOutputs(),
                                  nDefCount,
                                  m_pAnimController->GetMaxNumTracks(),
                                  m_pAnimController->GetMaxNumEvents(),
                                 &pNewController );
```

```
        // Release our old controller
        m_pAnimController->Release();

        // Store the new controller
        m_pAnimController = pNewController;

    } // End if too small!
```

Notice in the above code that when we clone the animation controller we feed in all the same controller limits as the current controller, with the exception of the second parameter. This is where we pass in the number of animation sets we will need this new controller to handle. Therefore, the new controller will inherit all the limits of the original controller but with an increase in the number of animation sets that can be registered. The original controller is then released and the actor's controller member pointer assigned the address of our cloned controller.

We will now set up a loop to process each split definition in the passed array. For each one, we will fetch the index of the source set it references. This is the index of the animation set from which the keyframe data will be extracted. If this is not an ID3DXKeyframedAnimationSet then we will skip this definition. Provided it is a keyframed animation set, we create a new ID3DXKeyframedAnimationSet. The name of this set is the name that was specified by the application and passed in the split definition.

```
    // We're now going to start building for each definition
    for ( i = 0; i < nDefCount; ++i )
    {
        // First retrieve the animation set
        if ( FAILED(m_pAnimController->GetAnimationSet( pDefList[i].SourceSet,
                                                        &pAnimSet )) ) continue;

        // We have to query this to determine if it's keyframed
        if ( FAILED( pAnimSet->QueryInterface( IID_ID3DXKeyframedAnimationSet,
                                               (LPVOID*)&pSourceAnimSet ) ) )
        {
            // Release animation set and continue
            pAnimSet->Release();
            continue;

        } // End if failed to retrieve keyframed set

        // We can release the original animation set now, we've 'queried' it
        pAnimSet->Release();

        // Create a new 'destination' keyframed animation set
        D3DXCreateKeyframedAnimationSet(pDefList[i].SetName,
                                        pSourceAnimSet->GetSourceTicksPerSecond(),
                                        pSourceAnimSet->GetPlaybackType(),
                                        pSourceAnimSet->GetNumAnimations(),
                                        0,
                                        NULL,
                                        &pDestAnimSet );
```

Look at the call to the D3DXCreateKeyframedAnimationSet function above. With the exception of the first parameter, where we pass in a new name, we inherit all other properties from the source set (playback type, number of animations, and source ticks per second ratio).

With our new animation set created, our next task is to loop through each animation in the source animation set and extract the scale, rotation, and translation keys into temporary arrays. The following code fetches the SRT key counts for the animation currently being processed. It then allocates SRT arrays to store a copy of the source SRT data. Then, it stores a copy of the SRT key data in the temporary SRT arrays so that we have a complete copy of the keyframe data to work with. Notice that one of the first things we do is fetch the name of the animation we are processing. We will need this later when we wish to register the copy of the animation with the new animation set. We need to make sure that the animation names remain intact since this provides the mapping between the animation data and the frame in the hierarchy it animates.

```
        // Loop through the animations stored in the set
        AnimationCount = pSourceAnimSet->GetNumAnimations();
        for ( j = 0; j < AnimationCount; ++j )
        {
            // Get name
            pSourceAnimSet->GetAnimationNameByIndex( j, &strName );

            // Retrieve all the key counts etc
            ScaleKeyCount           = pSourceAnimSet->GetNumScaleKeys( j );
            RotationKeyCount        = pSourceAnimSet->GetNumRotationKeys( j );
            TranslationKeyCount     = pSourceAnimSet->GetNumTranslationKeys( j );
            NewScaleKeyCount        = 0;
            NewRotationKeyCount     = 0;
            NewTranslationKeyCount  = 0;

            // Allocate enough memory for the keys
            if ( ScaleKeyCount )
                ScaleKeys = new D3DXKEY_VECTOR3[ ScaleKeyCount ];

            if ( TranslationKeyCount )
                TranslateKeys = new D3DXKEY_VECTOR3[ TranslationKeyCount ];

           if ( RotationKeyCount )
                RotateKeys = new D3DXKEY_QUATERNION[ RotationKeyCount ];

            // Allocate enough memory (total) for our new keys,
            // + 2 for potential start and end keys
            if ( ScaleKeyCount )
                  NewScaleKeys = new D3DXKEY_VECTOR3[ ScaleKeyCount + 2 ];

           if ( TranslationKeyCount )
                  NewTranslateKeys = new D3DXKEY_VECTOR3[TranslationKeyCount+ 2 ];

           if ( RotationKeyCount )
                  NewRotateKeys = new D3DXKEY_QUATERNION[ RotationKeyCount + 2 ];

            // Retrieve the physical keys
            if ( ScaleKeyCount )
               pSourceAnimSet->GetScaleKeys( j, ScaleKeys );
```

```
            if ( TranslationKeyCount )
               pSourceAnimSet->GetTranslationKeys( j, TranslateKeys );

            if ( RotationKeyCount )
                 pSourceAnimSet->GetRotationKeys( j, RotateKeys );
```

The inner loop is concerned with copying the keyframe data for a single animation in the source set into a single animation in the destination set. Further, we will only copy over keyframes that fall within the start and end markers specified in the definition. What we must consider however is that the definition may have a start or end marker set between two keyframes in the source set. For example, imagine an animation has two translation keys at 0 and 10 seconds respectively. It is possible that we have placed a start and end marker at 3 and 8 seconds, respectively. Since the only two keyframes that exist fall outside these two markers, we would end up not copying anything into the new animation set. This is certainly not the way it should work.

In such a case, if the start and end markers do not fall on pre-existing keyframe positions, we must generate them ourselves and insert them into the new animation set. In the current example, this means we would need to generate a start and an end keyframe to be inserted at 3 and 8 seconds and. These are the two keyframes that the final animation would contain.

So it seems that we might have a situation where the start or end marker is not situated on pre-existing keyframe positions, and when this is the case we may need to insert a new start or end keyframe (or both, if both markers are between existing keyframes). But how do we generate a scale, rotation and translation key for these ‘in-between’ times?

When you think about this problem, you realize that we are asking a question we have already answered numerous times before. The ID3DXKeyframedAnimationSet::GetSRT method does just that. All we have to do is pass it the start marker position and it will interpolate the correct scale, rotation and translation keys for those in-between times. In the next section of code, we call the GetSRT method twice to generate SRT keys for the StartTicks and EndTicks positions specified in the current definition. Because the GetSRT method expects a time in seconds and we have our split definition marker values in ticks, we must pass in the Start and End marker positions divided by the source ticks per second value of the animation set.

```
            // Get the SRT data at the start and end ticks
            // (just in case we need to insert them)
            pSourceAnimSet->GetSRT( (double)pDefList[i].StartTicks /
                                     pSourceAnimSet->GetSourceTicksPerSecond(), j,
                                    &StartScale, &StartRotate, &StartTranslate );

            pSourceAnimSet->GetSRT( (double)pDefList[i].EndTicks /
                                     pSourceAnimSet->GetSourceTicksPerSecond(), j,
                                    &EndScale, &EndRotate, &EndTranslate );

            // Swap 'winding', we're adding these directly backto the anim
            // controller, not using directly.
            D3DXQuaternionConjugate( &StartRotate, &StartRotate );
            D3DXQuaternionConjugate( &EndRotate, &EndRotate );
```

We may not actually need the two sets of SRT keys we have just generated. If the start and end markers fall on existing keyframe positions then we can simply copy them over into the new animation. However, if this is not the case, then we require the above step to interpolate the keys for the start and end markers. Notice that we swap the handedness of the quaternions returned so they are left-handed.

Now it is time to start copying over the relevant data from the source animation into the new animation. We will do this in three passes.

In this first pass we copy over the scale keys. In this first section of the scale key pass, we loop through each scale key in the array and skip any keys which have timestamps prior to the StartTicks position specified in the current split definition being processed. If a key has a smaller timestamp we simply continue the next iteration of the loop to process the next scale key. If the timestamp of the current scale key is larger than the EndTicks time (also supplied in the split definition structure) then it means that we have processed all scale keys between the start and end marker positions and we are not interesting in copying any keys after this point. When this is the case, we break from the loop and our job is done.

```
            // ******************************************************************
            // * SCALE KEYS
            // ******************************************************************
            for ( k = 0; k < ScaleKeyCount; ++k )
            {
                D3DXKEY_VECTOR3 * pScale = &ScaleKeys[k];

                // Skip all keys prior to the start time
                if ( pScale->Time < pDefList[i].StartTicks ) continue;

                // Have we hit a key past our last time?
                if ( pScale->Time > pDefList[i].EndTicks ) break;
```

If we get here then we have a key that needs to be copied. However, if we have not yet added any scale keys then this is the first. Also, if the timestamp of this first key we are about to add is not exactly the same as the StartTicks marker position stored in the definition, it means that the first keyframe we are about to copy occurs after the start tick position. We need a keyframe to be inserted at the start marker position so we insert the start scale key we calculated above using the GetSRT method. This will make this start key the first in the array (now definitively located at the start marker position). We then copy over the current key we are processing.

```
                // If we got here we're within range.
                //If this is the first key, we may need to add one
                if(NewScaleKeyCount==0 && pScale->Time != pDefList[i].StartTicks )
                {
                    // Insert the interpolated start key
                    NewScaleKeys[ NewScaleKeyCount ].Time  = 0.0f;
                    NewScaleKeys[ NewScaleKeyCount ].Value = StartScale;
                    NewScaleKeyCount++;

                } // End if insert of interpolated key is required

                // Copy over the key and subtract start time
                NewScaleKeys[ NewScaleKeyCount ] = *pScale;
```

```
            NewScaleKeys[ NewScaleKeyCount ].Time -= pDefList[i].StartTicks;
            NewScaleKeyCount++;
        } // Next Key
```

At the end of this loop we will have copied over all scale keys that existed in the original animation between the start and end marker positions. Further, we may have inserted a new start key at the beginning of the array if the first key we intended to copy had a greater timestamp than the starting marker position. We must always have keyframes on the start and end marker positions.

Now we must test to see if the last scale key we added in the above loop has a timestamp equal to the EndTicks position. If so, then all is well because the last key we copied is exactly where this animation set's timeline should end. However, if this is not the case, then it means the last scale key we copied occurred before the EndTicks position. In that case, we must insert an additional key at the end marker position. We calculated that end key ealier using GetSRT, so we add it to the end of the array.

```
        // Last key matched end time?
        if ( NewScaleKeys[NewScaleKeyCount - 1].Time !=
             pDefList[i].EndTicks - pDefList[i].StartTicks )
        {
            // Insert the interpolated end key
            NewScaleKeys[ NewScaleKeyCount ].Time  = pDefList[i].EndTicks –
                                                     pDefList[i].StartTicks;
            NewScaleKeys[ NewScaleKeyCount ].Value = EndScale;
            NewScaleKeyCount++;

        } // End if insert of interpolated key is required
```

That is the end of the first pass. We now have a temporary array called NewScaleKeys which contains all scaling information between the two markers specified by the split definition for the current animation being copied.

In the second pass we will do exactly the same thing, only this time we will be copying over translation keys instead. The code is nearly identical.

```
        // ******************************************************************
        // * TRANSLATION KEYS
        // ******************************************************************
        for ( k = 0; k < TranslationKeyCount; ++k )
        {
            D3DXKEY_VECTOR3 * pTranslate = &TranslateKeys[k];

            // Skip all keys prior to the start time
            if ( pTranslate->Time < pDefList[i].StartTicks ) continue;

            // Have we hit a key past our last time?
            if ( pTranslate->Time > pDefList[i].EndTicks ) break;

            // If we got here we're within range.
            // If this is the first key, we may need to add one
            if (NewTranslationKeyCount==0 &&
               pTranslate->Time!= pDefList[i].StartTicks )
```

```
                {
                    // Insert the interpolated start key
                   NewTranslateKeys[NewTranslationKeyCount].Time  = 0.0f;
                   NewTranslateKeys[NewTranslationKeyCount].Value =
                                                              StartTranslate;
                   NewTranslationKeyCount++;
                } // End if insert of interpolated key is required

                // Copy over the key and subtract start time
              NewTranslateKeys[NewTranslationKeyCount] = *pTranslate;
              NewTranslateKeys[NewTranslationKeyCount].Time -=
                                                    pDefList[i].StartTicks;
              NewTranslationKeyCount++;

            } // Next Key

            // Last key matched end time?
            if ( NewTranslateKeys[NewTranslationKeyCount - 1].Time !=
                 pDefList[i].EndTicks - pDefList[i].StartTicks )
            {
                // Insert the interpolated end key
                NewTranslateKeys[ NewTranslationKeyCount ].Time =
                              pDefList[i].EndTicks - pDefList[i].StartTicks;

              NewTranslateKeys[ NewTranslationKeyCount ].Value = EndTranslate;
                NewTranslationKeyCount++;

            } // End if insert of interpolated key is required
```

Once again, the code shows that if there are no keyframes existing in the source animation set on the start and end marker positions, we insert the start and end keys interpolated using GetSRT prior to the copy process.

At this point, two of the three copy passes are complete. We have two arrays that contain scale keys and translation keys that existed either on or between the two markers. In the final pass, we copy over the rotation information using the same strategy. Once again, we insert the start and end quaternion keys we interpolated earlier if it is necessary to use them.

```
            // ******************************************************************
            // * ROTATION KEYS
            // ******************************************************************
            for ( k = 0; k < RotationKeyCount; ++k )
            {
                D3DXKEY_QUATERNION * pRotate = &RotateKeys[k];

                // Skip all keys prior to the start time
                if ( pRotate->Time < pDefList[i].StartTicks ) continue;

                // Have we hit a key past our last time?
                if ( pRotate->Time > pDefList[i].EndTicks ) break;

                if ( NewRotationKeyCount == 0 &&
                   pRotate->Time != pDefList[i].StartTicks )
                {
```

```
                // Insert the interpolated start key
                NewRotateKeys[ NewRotationKeyCount ].Time  = 0.0f;
                NewRotateKeys[ NewRotationKeyCount ].Value = StartRotate;
                NewRotationKeyCount++;

            } // End if insert of interpolated key is required

            // Copy over the key and subtract start time
            NewRotateKeys[ NewRotationKeyCount ] = *pRotate;
            NewRotateKeys[ NewRotationKeyCount ].Time -=
                                                pDefList[i].StartTicks;
            NewRotationKeyCount++;

        } // Next Key

        // Last key matched end time?
        if ( NewRotateKeys[NewRotationKeyCount - 1].Time !=
           pDefList[i].EndTicks - pDefList[i].StartTicks )
        {
            // Insert the interpolated end key
            NewRotateKeys[ NewRotationKeyCount ].Time  =
                                pDefList[i].EndTicks - pDefList[i].StartTicks;

            NewRotateKeys[ NewRotationKeyCount ].Value = EndRotate;
            NewRotationKeyCount++;

        } // End if insert of interpolated key is required
```

With the copy process complete, we now have three arrays that describe the SRT data for one of the animations in our newly created animation set. Next we will register that SRT data with our new animation set. Notice that we supply the same name for the animation we are adding to the new animation set as the animation we are copying from. We retrieved this information previously at the start of the animation loop. We must make sure that when we register this new animation copy with our new animation set that we give it the same name as the animation it was copied from. This name provides the mapping between the SRT data contained in the animation and the hierarchy frame which it animates. Once we have added the new animation to our new animation set, we can release the temporary arrays we allocated for the copy process.

```
        // Register this with our destination set
        pDestAnimSet->RegisterAnimationSRTKeys(strName,
                                               NewScaleKeyCount,
                                               NewRotationKeyCount,
                                               NewTranslationKeyCount,
                                               NewScaleKeys,
                                               NewRotateKeys,
                                               NewTranslateKeys, NULL );

        // Clean up
        if ( ScaleKeys )   delete []ScaleKeys; ScaleKeys = NULL;
        if ( TranslateKeys )    delete []TranslateKeys; TranslateKeys = NULL;
        if ( RotateKeys )       delete []RotateKeys; RotateKeys = NULL;
        if ( NewScaleKeys )     delete []NewScaleKeys; NewScaleKeys = NULL;
```

```
            if ( NewRotateKeys )    delete []NewRotateKeys; NewRotateKeys = NULL;
            if ( NewTranslateKeys )
               delete []NewTranslateKeys; NewTranslateKeys = NULL;

        } // Next Animation
```

The animation copy loop we have just examined is executed for every animation in the source set. This creates corresponding (albeit edited) versions of those same animations in the new set. Once we have exited the inner loop, we have copied over all required animation data and have successfully dealt with the current split definition being processed. We have created a new animation set for it and have populated it with updated animation data.

Before moving on to processing the next split definition in the array, we release the interface to the original animation set. This does not delete it from memory since the controller is currently still using it. Our new animation sets will not be added to the controller until later, so for the time being, the controller still contains the original sets. In fact, the last thing we would want to do at this point is actually remove the source animation set from memory. Remember, we may have more split definitions to process and some of them may (and probably will) use the same source animation set for the copying process. So we will still need access to it. This will certainly be the case when the X file contains a single animation set containing all the animation sequences. We are only releasing the copy of the interface pointer that we were returned from pAnimationController->GetAnimationSet( ) at the start of the split definitions loop. The outer loop (i.e., the split definition loop) closes like so:

```
        // We're done with the source 'keyframed' animation set
        pSourceAnimSet->Release();

        // Store the dest animation set for use later
        pDefList[i].Reserved = (LPVOID)pDestAnimSet;

    } // Next Definition
```

Notice that this function uses the reserved member of the split definition structure it has just processed to store a temporary pointer to the new animation set it just created. Now we see why this member was reserved -- it gives the function a temporary place to store the animation sets created for each split definition until we register them with the controller later on.

At this point in the function, we have processed all split definitions and have created a new animation set for each. We will now need to register these animation sets with the controller. But before we do, we first need to remove any existing sets. We no longer need these old animation sets as they were merely the source data containers for the new animation sets we have created.

```
    // Release all animation sets from the animation controller
    // (Note: be careful because the GetNumAnimationSets result,
    // and the values passed in to GetAnimationSet alter ;)
    for ( ; m_pAnimController->GetNumAnimationSets() > 0; )
    {
        // Retrieve the animation set we want to remove
        m_pAnimController->GetAnimationSet( 0, &pAnimSet );
        // Unregister it (this will release inside anim controller)
        m_pAnimController->UnregisterAnimationSet( pAnimSet );
```

```
        // Now release the last known copy
        pAnimSet->Release();

    } // Next Animation Set
```

We now have an empty animation controller and it is time to register our new animation sets. Remember that the Reserved member of each split definition structure was used to store an interface pointer to the animation set that we created for that definition. So all we have to do now is loop through each split definition and register the animation sets stored there with the controller.

```
    // Now add all the new animation sets back into the controller
    for ( i = 0; i < nDefCount; ++i )
    {
        // Retrieve back from the structure
        pDestAnimSet = (LPD3DXKEYFRAMEDANIMATIONSET)pDefList[i].Reserved;
        if ( !pDestAnimSet ) continue;

        // Register with the controller (addrefs internally)
        m_pAnimController->RegisterAnimationSet( pDestAnimSet );

        // Release our local copy and clear the list item just in case
        pDestAnimSet->Release();
        pDefList[i].Reserved = NULL;

    } // Next Definition
```

Now we have an animation controller with new animation sets defined by the split definition array we passed in. Since we have removed all the old animation sets, there will not currently be an animaton set assigned to any track. Therefore, this function will perform a default assignment of the first set to track 0 on the mixer.

```
    // Set the first set in track 0 for default
    m_pAnimController->GetAnimationSet( 0, &pAnimSet );
    m_pAnimController->SetTrackAnimationSet( 0, pAnimSet );
    pAnimSet->Release();

    // Success!!
    return D3D_OK;
}
```

That was a fairly large function and one you will probably not call directly very often. As discussed, Lab Project 10.2 is a splitter tool which is essentially built entirely around this one function call. It provides a nice way around a problem for people who have their animation data combined in the X file and wish to divide it into separate sets. ApplySplitDefinitions is a utility function and should not be used in a time critical portion of your code. Also note that if you intend to save the actor back out to an X file, then you must make sure that the actor is created in managed mode. Finally, if you wish to use this function, you will want to disable the call to ApplyCustomSets in CActor::LoadActorFromX since this function expects to work with the data extracted from the X file. It has no concept of callback keys (as they are never specified in an X file) and it has no concept of our CAnimationSet object either.

Supporting CActor Instancing

Although we have finished adding animation support to the CActor class, we will use this lab project to add some much needed support to our applications for the correct instancing of animated actors. As we know from previous lessons, the ability to have several objects in our scene that have unique positions but share the same mesh data is vitally important. Instancing allows us to avoid storing multiple copies of identical geometry (and other properties) in memory. Indeed, we have used mesh instancing going back all the way to some of the first projects we have done together. This of course is why our CScene class represents objects using the CObject class. The CObject class contains a world matrix that describes the position of the object in the world. It also has a pointer to a separate mesh or actor which points to the actual model space data of the object's associated mesh(s).

In the case of a simple mesh, instancing is easy. If we have a mesh of a street lamp that we wish to place at multiple locations in the scene, we can load the street lamp mesh once and then have several CObject's reference that mesh. Each CObject contains its own world matrix that can be used to position the mesh in the correct place in the scene. In the case of simple static meshes like this, all we have to do before we render an object is set its world matrix on the device and then render the mesh it points to. Even if several CObject's point to the same mesh, as long as we set each object's world matrix prior to rendering its associated mesh, the pipeline will correctly transform and render the mesh in the correct position in the scene. Since this is how we render non-instanced objects as well, our application does not care whether it is the only object referencing that mesh data or not. It all works the same way.

When dealing with actors, instancing support is not quite so automatic or free from CPU overhead. An actor contains a hierarchy of matrices which must all be updated to position the actor in the scene using the associated object's world matrix. Unlike a single mesh object where we can just set the object's matrix as the current world matrix on the device and then render that mesh, when we set the world matrix of an object that stores an actor, we do not simply set the object's matrix as the device world matrix. In fact, we use the CActor::SetWorldMatrix function, passing in the object's world matrix. This function will multiply that world marix with the root frame's relative matrix, changing the root frame of reference for the entire hierarchy. It will then need to perform a full traversal of the hierarchy to rebuild the absolute world matrices stored at each frame. Once done, each frame in the hierarchy will contain its absolute world matrix and thus any mesh attached to that frame will have the matrix it needs to set on the device prior to being rendered.

After the hierarchy matrix update to world space, the rendering step involves nothing more than traversing the hierarchy and searching for mesh containers. When a mesh container is found, the absolute matrix stored in the owner frame will be the world matrix that must be set on the device prior to rendering that mesh. As long as we call the CActor::SetWorldMatrix function and pass in the object's world matrix prior to rendering that actor, we will update the matrices of the actor that describe where the mesh components comprising the actor should be positioned in the scene (using the object's matrix as a frame of reference). If multiple objects reference the same actor, it no longer matters. As long as we update the matrices of the actor using each object's world matrix prior to rendering it.
Of course, we have to be aware of the potential overhead that comes with referencing an actor, even though this same overhead would be incurred if we loaded five unique actors. For example, if we have five CObject's that point to the same actor, then we will have to set the world matrix for the actor five times, once for each time we render the actor for a given object. This means five hierarchy traversals

will be peformed to update the matrices of the actor such that it can be rendered in five different locations. However, even if we used five unique actors, we would still need to traverse each actor's hierarchy every time its associated object world matrix was updated. Of course, this is where the subtle difference comes into play between using five separate actors and referencing the same actor five times.

In the case when five unique actors are being used, each actor would only have to have its hierarchy updated if its associated world matrix had been updated. For example, if only one of the objects had its matrix updated, the other four actors could be left alone since their matrices are currently up to date. Only the actor whose world matrix has been updated would need to have all its matrices rebuilt before being rendered. In the case of one actor being referenced by five different objects, even if none of those objects have had their world matrices changed since the last update, the matrices of the actor will always have to be rebuilt five times, once for each object that uses it. This is because we must use the actor's matrices to take a snapshot of the actor from multiple locations in the scene (once for each object). Thus we will always have to rebuild the matrices of the actor every frame for every object that references it. While this usually means many more hierarchy traversals than in the case of loading the geometry multiple times, the memory savings is significant and often the only practical solution.

> **Note:** It should be noted that a Clone function could be added to our CActor to create a new actor that is a copy. This new actor, while having its own unique hierarchy and matrices, could have mesh containers that point to the original mesh containers in the actor from which it was cloned. The result is the ability to have multiple CActor's that all share the same mesh data (model space vertices and indices) but that have their own unique hierarchies. This is like a hybrid of the non-referenced and referenced approaches. Memory would still be consumed by the multiply hierarchies maintained by each actor, and this might not always be practical, but the mesh data would be shared. With such a system, any actors that have not had their associated object's world matrix updated would not need to have their hierarchy matrices updated needlessly. We will leave the creation of such a function as an exercise for the reader. In our applications, we will be using classic referencing when more than one CObject points to a given actor. As such, there will be only one copy of an actor's hierarchy and mesh data in memory at any given time.

Instancing actors is quite simple if the actor is not animated. For example, the following code snippet shows how the objects could be rendered inside our CScene::Render function. We are assuming that managed meshes are being used to remove the setting of textures and materials from the listing and to simplify the point we are trying to make clear.

```
for ( i = 0; i < m_nObjectCount; ++i )
{
   CObject  * pObject = m_pObject[i];
   if ( !pObject ) continue;

   // Retrieve actor and mesh pointers
   CActor   * pActor = pObject->m_pActor;
   CTriMesh * pMesh  = pObject->m_pMesh;
   if ( !pMesh && !pActor ) continue;

   // Set up transforms
   if ( pMesh )
   {
      // Setup the per-mesh / object details
      m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i]->m_mtxWorld );
      m_pD3DDevice->SetFVF( pMesh->GetFVF() );
```

```
      // Draw Mesh (managed mode example)
      pMesh->Draw();

   } // End if mesh

   else

   {
      // Set world matrix and update combined frame matrices.
      pActor->SetWorldMatrix( &pObject->m_mtxWorld, true );

      // Draw Actor (Managed mode example)
      pActor->DrawActor()

   } // End if actor

} End for each object
```

As the above code shows, we loop through each object in the scene and fetch either its mesh or actor pointer. If the object contains a pointer to a mesh, then the object's world matrix is bound to the device and the mesh is rendered. If the object points to an actor, the CActor::SetWorldMatrix function is used. This function does not actually set any matrices on the device, it combines the passed world matrix with the root frame matrix, changing the frame of reference for the entire actor. The 'true' parameter that is passed instructs the function to then traverse the hierarchy and rebuild all the absolute matrices for each frame in the hierarchy. We then call CActor::DrawActor to render the managed mode actor. This function is the function that traverses the hierarchy and sets the matrices on the device prior to rendering each mesh contained there.

The above code works flawlessly whether actors or meshes are being referenced (or not). The code does not care if a CObject's associated mesh or actor is being used by more than one object because the same rendering logic works in both cases. If animation was not a factor, that is all we would have to do to provide correct instancing support for our actors. However, as we have learned in this chapter, an actor can also contain animation data and it is here that our current instancing method would break if we were to take no additional measures. In fact, the code we have used above was implemented in previous lessons so we have had support for static actor instancing for some time now. With animated actors things get a little more complicated and we will have to perform some additional logic and add two new member functions to CActor to make sure that animated actor instancing is properly done.

Who Controls the Controller?

When an actor contains animation data, we update the animations each frame by calling the actor's AdvanceTime method. This method passes the request on to the underlying animation controller which updates the relative matrices to reflect the changes caused by the advance in the timeline. As previously described, we do this inside the CScene::AnimateObjects function, and our current animation update strategy might look like the following:

```
void CScene::AnimateObjects( CTimer & Timer )
{
    ULONG i;
    static CActionHandler Handler;

    // Process each object for coll det
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CObject * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Get the actor pointer
        CActor * pActor = pObject->m_pActor;

        if ( !pActor ) continue;

        // Advance time
        pActor->AdvanceTime( Timer.GetTimeElapsed(), false, &Handler );

    } // Next Object
}
```

As you can see, the above code loops through each CObject in the scene and tests to see if that object uses an actor or a mesh. If the object uses a mesh, then there is nothing to do because a single mesh does not contain animation data. If the object uses an actor, then the actor's animation controller is advanced by calling the CActor::AdvanceTime method. If the actor has no animation data then this is a no-op and we can safely call it for all actors.

The problem happens when multiple CObject's are referencing the same actor. Since the actor houses the animation controller, if multiple objects reference the same actor, they are also referencing a shared animation controller. If we had five objects that all referenced the same actor, the above function would not advance the controller time of five actors, it would advance the time of a single actor's controller five times in a given frame update. Therefore, every object would share the same animaton data and the same position in the animation's timeline. Furthermore, that timeline would be playing back at an incorrect speed because we are advancing the timeline multiple times in a given update (once for each object referencing the actor). This will simply not do. Although we can certainly live with the fact (and will actually often desire it) that all objects that reference the actor will essentially contain the same animation data and will also have to have synchronized timelines, we certainly can not settle for those animatons playing back at faster speeds than intended. The situation obviously gets a lot worse if there were hundreds of ojects referencing that actor as its animation timeline would be updated hundreds of times in a given frame update.

We could work around such a problem by using a 'frame time' variable inside the actor so that any calls to AdvanceTime past the first that have identical frame times are ignored. That way, the actor would ignore any AdvanceTime calls except the first for a given frame update. If this technique were implemented, even if 100 objects called AdvanceTime for a single actor in a given frame update, only the first would be processed. But this is still far from ideal; the system we are going to use will provide much more flexibility. Not only will it allow each object to have independent timelines and animation speeds, it can also give each object that references a single actor its own unique animation data.

The root cause of our problem is that every object shares the same animation controller stored inside the actor they are referencing. However, if we were to change this relationship and instead give each object its own copy of the controller (cloning the original controller that was loaded with the actor data) we remove this conflict. We could update our CObject structure so that it now also maintains a pointer to an ID3DXAnimationController like so:

```
class CObject
{
public:
    //-----------------------------------------------------------------------
    // Constructors & Destructors for This Class.
    //-----------------------------------------------------------------------
    CObject( CTriMesh * pMesh  );
    CObject( CActor   * pActor );
    CObject( );
    virtual ~CObject( );

    //-----------------------------------------------------------------------
    // Public Variables for This Class
    //-----------------------------------------------------------------------
    D3DXMATRIX                  m_mtxWorld;             // Objects world matrix
    CTriMesh                   *m_pMesh;                // Mesh we are instancing
    CActor                     *m_pActor;               // Actor we are instancing
    LPD3DXANIMATIONCONTROLLER   m_pAnimController       // Instance Controller
};
```

Since each object has its own controller, it has the ability to move that timeline along at any speed and to whatever position it desires. As mentioned, we could conceivably even generate completely unique animation data for each object as long as the names of the frames comprising the actor's hierarchy are known. For this to work, before we update the animation time of each object, we would first need to attach it to the actor. We can then use the actor's animation functions to alter the timeline of the controller for that particular object. We may also want to allow the attaching of the controller to automatically synchronize the relative matrices of the hierarchy to the data contained in the controller. Simply storing the controller's pointer inside the actor will not update the hierarchy to reflect the current pose described by the timeline of that controller.

Our new version of CScene::AnimateObjects is shown below. It has an additional line of code that calls a CActor method yet to be implemented (AttachController). For now, do not worry about how the object is assigned its own animation controller; we will see this in a moment when examining the changes to the CScene IWF loading code. For now just know that if multiple CObject's reference the same animated actor, the actor will no longer contain its own animation controller -- the CObjects will have their own copy.

```
void CScene::AnimateObjects( CTimer & Timer )
{
    ULONG i;
    static CActionHandler Handler;

    // Process each object for coll det
    for ( i = 0; i < m_nObjectCount; ++i )
    {
```

```
        CObject * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Get the actor pointer
        CActor * pActor = pObject->m_pActor;
        if ( !pActor ) continue;

        if ( pObject->m_pAnimController )
            pActor->AttachController( pObject->m_pAnimController, false );

        // Advance time
        pActor->AdvanceTime( Timer.GetTimeElapsed(), false, &Handler );

    } // Next Object
}
```

As you can see, we still loop through each object searching for actors to advance their timelines. However, when we find an object that has its own animation controller we attach it to the actor. This function simply copies the passed animaton controller pointer into the actor's controller member variable (as it no longer contains its own animation controller). Once this has been done, we can then use the CActor::AdvanceTime function (and all the other CActor controller manipulation methods) to advance the timeline of the controller. Everytime we attach a new controller to the actor it overwrites the pointer of the controller that was previously stored.

The second parameter to the CActor::AttachController is a Boolean variable that dictates whether we would like the relative matrices of the actor's hierarchy to be updated to reflect the pose of the actor as defined by the current position of the controller's timeline. At this point we do not because we are simply using the actor interface to advance the timelines of each object's controller. You will see in a moment how we will call this function again in the CScene::Render function prior to rendering each CObject. That is, before we render each object in our world, if it is an actor, we will attach the object's controller to the actor, this time passing true as the Boolean parameter. This will cause the relative matrices of the actor to be synchronized to those described by the controller so that the actor is in the correct pose described by the currently attached controller. We will then set the actor's world matrix to initiate a traversal of the hierarchy and a rebuilding of the absolute matrices. These world matrices will be populated as described by the controller. We will then render the actor as we usually would. The snippet of rendering code in our CScene::Render function now looks something like this:

```
for ( i = 0; i < m_nObjectCount; ++i )
{
   CObject  * pObject = m_pObject[i];
   if ( !pObject ) continue;

   // Retrieve actor and mesh pointers
   CActor   * pActor = pObject->m_pActor;
   CTriMesh * pMesh  = pObject->m_pMesh;

   if ( !pMesh && !pActor ) continue;

   // Set up transforms
   if ( pMesh )
   {
```

```
      // Setup the per-mesh / object details
      m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_pObject[i]->m_mtxWorld );
      m_pD3DDevice->SetFVF( pMesh->GetFVF() );

      // Draw Mesh (managed mode example)
      pMesh->Draw();

   } // End if mesh

   else

   {
      // Attach controller with frame synchronization
      if ( pObject->m_pAnimController )
          pActor->AttachController( pObject->m_pAnimController, true )

      // Set world matrix and update combined frame matrices.
      pActor->SetWorldMatrix( &pObject->m_mtxWorld, true );

      // Draw Actor (Managed mode example)
      pActor->DrawActor()

   } // End if actor

} End for each object
```

As you can see, the only line added is the one that attached the controller of an object to the actor prior to rendering it.

Let us now examine the two new functions we will add to CActor that will allow us to attach and detach an animation controller.

CActor::AttachController

AttachController replaces the animation controller pointer currently stored in the actor with the one that is passed in. This is our means of connecting each CObject's controller to the actor so that we can use its methods to manipulate the timeline and update the actor's matrices for the given instance.

The function takes two parameters. The first is a pointer to the new ID3DXAnimationController that will be stored in the actor. Any previous controller stored in the actor will be released. If the controller currently stored in the actor is not being referenced reference elsewhere then it will be freed from memory. Usually, the controller stored here will also have an interface reference in the CObject that is referencing the actor, so in most cases we are simply releasing the actor's claim on that interface. The second parameter is a Boolean variable that controls whether the function should take the additional step of updating the relative matrices of the hierarchy so that they are synchronized to the timeline of the new controller.

As discussed previously, this function is called twice for each CObject that references an actor. The first time it is called (inside CScene::AnimateObjects) we pass false as the second parameter because we do not want to synchronize the matrices of the actor to the controller at this point. We simply want to attach

the controller so that we can use the actor interface to advance the timeline. The second time we call this method for each CObject is when we are about to render the object. This time, we attach the controller and pass true as the Boolean parameter so that the relative matrices are synchronized to the positions described by the animation data in the controller for the current position on the timeline. We then set the world matrix of the actor, which concatenates the relative matrices in their newly animated states, to build the absolute world matrices of the actor at each frame. Finally, we render the actor.

The first section of code releases the actor's claim on any controller interface that is currently being referenced and sets the controller pointer to NULL.

```
void CActor::AttachController( LPD3DXANIMATIONCONTROLLER pController,
                               bool bSyncOutputs /* = true */ )
{
    // Release our current controller.
    if ( m_pAnimController ) m_pAnimController->Release();
    m_pAnimController = NULL;
```

The next piece of code is all in a conditional block that is only executed if the first parameter to the function is not NULL. An application could pass NULL as the first parameter instead of a valid pointer to an animation controller. The result would be the removal of the controller currently being used by the actor and the setting of this pointer to NULL.

Provided the application has passed a valid controller pointer the next code block will be executed. It stores the controller poiner in the actor member variable and increases its reference count. If the second parameter to the function was false, then our job is done. However, if true was passed, then we want the controller we have just attached to update the relative matrices of the hierarchy.

How does this happen? Well, we know that when we advance the time of the animation controller, this will automatically cause the controller to update the relative matrices (animation outputs) of the hierarchy it is animating. However, we do not want to physically advance the time of the controller. That is not a problem. All we have to do is call AdvanceTime and specify that we would like the timeline advanced by zero seconds. The remainder of the function is shown below:

```
    if ( pController )
    {
        // Store the new controller
        m_pAnimController = pController;

        // Add ref, we're storing a pointer to it
        m_pAnimController->AddRef();

        // Synchronise our frame matrices if requested
        if ( bSyncOutputs ) m_pAnimController->AdvanceTime( 0.0f, NULL );

    } // End if controller specified
}
```

By calling AdvanceTime and passing in zero seconds, we do not erroneously alter the position of the timeline (after all, we do that in CScene::AnimateObjects). However, calling this function does instruct the controller to fetch the SRT data for each matrix for the current the position on the timeline. This SRT

data is retrieved from the controller's animation data and used to rebuild the relative frame matrices. We will see in a moment when we examine the loading code that while the CObjects now own their own controllers, these controllers are all clones of the original controller that was created when the actor was first loaded from the X file. Therefore, all the controllers will have correctly had all the frames in the hierarchy that require animation registered as animation outputs.

The next function we will add to the CActor class is called DetachController and is the mirror of the current function. It just sets the controller pointer of the actor to NULL and returns the controller interface currently being used by the actor back to the application. After this function has been called, the actor's internal controller pointer will be set to NULL.

Before we discuss why the application might wish to fetch the controller currently being used by the actor and also have it removed from the actor at the same time, let us look at the function code.

CActor::DetachController

DetachController returns an interface to the controller currently being used by the actor and sets the controller pointer member variable to NULL. As such, the actor has no controller attached after this call. Here is the code:

```
LPD3DXANIMATIONCONTROLLER CActor::DetachController( )
{
    LPD3DXANIMATIONCONTROLLER pController = m_pAnimController;

    // Detach from us
    m_pAnimController = NULL;

    // Just return interface
    return pController;

}
```

The function takes no parameters and returns an ID3DXAnimationController interface pointer back to the caller. Normally, when we remove an object's claim on an interface we would call Release on that interface before setting the pointer to NULL. Notice that we do not do so in this case -- we simply set the pointer to NULL. This is because under normal circumstances, when a function returns an interface pointer, it would increase the reference count of the interface before returning that pointer. In this case, it is unnecessary. Normally we would first increase the reference count when we make the interface pointer copy that will be returned back to the application and would then call Release on that interface before setting the actor's pointer to NULL. Since these two steps cancel each other out, we will simply set the actor pointer to NULL and transfer ownership of the interface reference to the function caller. As the actor loses the reference to that interface, the caller will gain it. The calling function should call Release on this interface when it no longer wishes to use it.

In order to understand why we might need to detach a controller, we need to understand the loading process. That is what we will discuss next.

In our current applications, the only way multiple instances of an actor (or even multiple actors) can be created is if we are loading the scene file from an IWF file using CScene::LoadSceneFromIWF. This is because the alternative method of scene loading we provide (CScene::LoadSceneFromX) assumes that the entire scene is stored in a single X file and thus, only one actor will ever be created. In the case of IWF loading, the situation is different. While the IWF file may contain several internal meshes, these meshes will be loaded and stored as single mesh CObjects in the scene. We can think of these internal meshes as the brushes that you might place in GILES™ since this geometry is stored directly in the IWF file and can contain no animation or hierarchy data. The IWF function that picks these internal meshes out of the IWF file during the loading process is the CScene::ProcessMeshes function. For each mesh found in the IWF file, its data will be copied into a new CTriMesh object and stored in a new CObject. Our actor class is not used for static meshes stored in the IWF file.

However, we do use actors to load reference entities that are stored in the IWF file. A reference entity in an IWF file is an entity that contains a world matrix and a filename. This filename is the name of an X file that contains an object or object hierarchy positioned in your scene using a tool like GILES™. There are many benefits to using references in this way.

For example, imagine you have a very nice X file which contains the representation of a tree. This tree might be comprised of several meshes stored in a hierarchy inside the X file. Furthermore, this X file might also contain animation data that has been hand crafted to allow the tree to sway back and forth. If you were to import this X file into GILES™ as a regular mesh, the seperate meshes comprising the tree would all be collapsed into a single mesh and the hierarchy and animation data would be lost (as GILES™ has no native support for them). This is most likely not the outcome you want.

A better approach would be to design your animated X file in a package such as 3D Studio MAX™ or Maya™ and export the data out to an X file. You can then place multiple references to this X file in the scene using the GILES™ reference entity. For example, you might place 100 of your trees in the level. The geometry of these trees will never be stored in the IWF file. All that will be stored in the IWF file when you save the scene in GILES™ is the position and orientation of each reference (its world matrix) and the name of the X file that should be loaded by the application. This is helpful because it allows you to design your individual objects in your favorite modeling packages and use GILES™ as a simple scene placement tool. You can even create an IWF file in GILES™ which has no internal geometry whatsoever. It might just be a collection of reference entities describing where these external X file objects should be positioned in the scene.

When parsing reference entities during the loading of an IWF file our actor class will be used. The CScene::ProcessReference function is called during the loading process to extract the reference entity information. For each entity loaded we will create a new CObject. There will be a world matrix (describing its position in the scene) for that reference that we will store in the CObject's world matrix. The filename can be passed into a new CActor using CActor::LoadActorFromX as we have done in past lessons. This actor pointer will then be stored in the new CObject and added to the scene's CObject array.

None of this is new to us, but the next part is. When the LoadActorFromX method returns, the actor will have been fully populated with hierarchy data and an animation controller (if animation existed in the X file). In the past, we have simply assumed that an actor will never be referenced by more than one object

and as such, it was fine for the actor to own the animation controller. We now know however, that if more than one CObject is referencing the actor, the actor should no longer own the controller. Instead, each CObject will have its own copy of the original controller. (If an actor is only referenced by one CObject, then we should just let the actor own the controller and we can set the CObject's controller to NULL.)

Essentially, we have created two operating modes for CActor. If only one CObject references it, it will be allowed to own the controller it was originally assigned in the CActor::LoadActorFromX function. However, as soon as we discover that we are about to create an object that references an actor we have already loaded, we will detach the controller from the actor and store it in the CObject. For every CObject created that references that same actor, we will clone this controller and assign it to the CObject's controller pointer.

See this in action will make it much easier to understand, so let us now have a look at the updated code to the CScene::ProcessReference function. This is the function that is called during the loading of an IWF file to extract the information for each reference entity found in the file. This function is called from CScene::ProcessEntities since references are a type of IWF entity.

CScene::ProcessReference

This function is called by the CScene::ProcessEntities function each time a reference entity is encountered. The function is passed a ReferenceEntity structure which contains information about the reference (filename, etc.), and a reference world matrix describing where it should be positioned in the scene.

The first thing we do is test to see that the ReferenceType member of the ReferenceEntity structure is set to 1. If it is not, we return. A setting other than 1 indicates that this is an internal reference instead of an external reference. Internal references are not supported by our applications at this time.

```
bool CScene::ProcessReference( const ReferenceEntity& Reference,
                               const D3DXMATRIX & mtxWorld )
{
    HRESULT                     hRet;
    CActor                    * pReferenceActor      = NULL;
    LPD3DXANIMATIONCONTROLLER   pController          = NULL;
    LPD3DXANIMATIONCONTROLLER   pReferenceController = NULL;

    // Skip if this is anything other than an external reference.
    if (Reference.ReferenceType != 1) return true;
```

We will now extract the filename of the X file we need to load from the ReferenceName member of the input structure. As this will contain just the filename (not the complete path) we will need to append this filename to the data path string currently being used by the application. The CScene::m_strDataPath member is a string that contains the current folder where application data is stored (e.g., '\MyApp\MyData\'). This is the folder where our application expects to find textures and X file resources.

```
    // Build filename string
    TCHAR Buffer[MAX_PATH];
    _tcscpy( Buffer, m_strDataPath );
    _tcscat( Buffer, Reference.ReferenceName );
```

At this point we have the complete filename of the X file we wish to load (including path) stored in the local Buffer variable.

We now loop through every actor that is currently in the scene's CActor array and test to see if an actor already exists with the same name. If so, then it means that we have already loaded this X file for a previous actor and do not wish to load it again.

```
    // Search to see if this X file has already been loaded
    for ( ULONG i = 0; i < m_nActorCount; ++i )
    {
        if (!m_pActor[i]) continue;
        if ( _tcsicmp( Buffer, m_pActor[i]->GetActorName() ) == 0 ) break;

    } // Next Actor
```

At this point, if loop variable 'i' does not contain the same value as m_ActorCount then it means the above loop exited prematurely in response to finding an existing actor created from the same X file. When this is the case, we do not want to load the X file again; instead we will create a new CObject that references the current actor. However, there are some things we now wish to consider.

We will first need to find the CObject that currently contains a pointer to this actor and test to see if its animation controller pointer is NULL. If it is, then it means that the object is the only one that currently references the actor and at present, the actor owns its own controller. When this is the case, we will detach the controller from the actor and store it in that object instead. We will then clone this controller and store it in the object reference we are about to create. This is essentially the code that recognizes when more than one CObject is referencing an actor and removes the controller management responsibilities from the actor in favor of the object.

Next we see the code that executes this step. First we get a pointer to the actor that our new object is going to reference and then get a pointer to the pre-existing CObject that currently owns it.

```
    // If we didn't reach then end, this Actor already exists
    if ( i != m_nActorCount )
    {
        // Store reference Actor.
        pReferenceActor = m_pActor[i];

        // Find any previous object which own's this actor
        for ( i = 0; i < m_nObjectCount; ++i )
        {
            if (!m_pObject[i]) continue;
            if ( m_pObject[i]->m_pActor == pReferenceActor ) break;

        } // Next Object
```

Notice that the above code exits as soon as any CObject that currently references the actor is found. If multiple CObjects already exist at this point, and they all reference the same actor, then it does not matter which one we use. We are only going to use it to clone its controller into the new CObject reference we are about to create. If however, the object that we find has its animation controller pointer set to NULL then we know that is currently the only CObject that exists that currently references the actor. As such, the actor currently manages the controller. In this case we will detach the controller and assign it to this object. We will then clone this controller and assign the clone to the new object reference we are about to create.

The next code block is executed if an object was found in the above loop. (This should always be the case since there is currently no way that a CActor can be created that is not also assigned to a CObject.) First we get a pointer to that object and test to see if its animation controller pointer is set to NULL. If so then we know it is currently the only object that references this actor and therefore the actor currently contains the controller. As we are just about to create a new object reference to this actor we know the actor mode needs to change. Therefore, we will detach the controller from the actor and store it in the pre-existing object that we found.

```
        // Did we find an object?
        if ( i != m_nObjectCount )
        {
            CObject * pReferenceObject = m_pObject[i];

            // Is this the first reference?
            if ( !pReferenceObject->m_pAnimController )
            {
                // If this is the first time we've referenced this actor then
                // we need to detach it's controller and store in the reference
                // object
                pReferenceObject->m_pAnimController =
                                            pReferenceActor->DetachController();

            } // End if first reference
```

We have just switched the previous object from non-reference mode to reference mode by assigning it its own controller. At this point, the actor no longer owns a controller.

Our next step is to create the new object reference we actually entered this function to build. We know that the reference we are processing at this point references an actor that we already have in memory, so we just have to create a new CObject and add a pointer to the actor. We also know that this is not the only object that references the actor, so this new object will need to have its own animation controller as well.

With the previous object reference taken care of, we will prepare the data for the new object reference we are about to create. Our new object will need its own animation controller, so we will clone the animation controller that we just detached from the actor and assigned to the previous object. As the clone function requires that we pass in all the limitations of the new controller at cloning time, we will fetch these limits from the controller that we are about to clone prior to calling the clone method.

```
            ULONG nMaxOutputs, nMaxTracks, nMaxSets, nMaxEvents;
```

```
            // Retrieve all the data we need for cloning.
            pController = pReferenceObject->m_pAnimController;
            nMaxOutputs = pController->GetMaxNumAnimationOutputs();
            nMaxTracks  = pController->GetMaxNumTracks();
            nMaxSets    = pController->GetMaxNumAnimationSets();
            nMaxEvents  = pController->GetMaxNumEvents();

            // Clone the animation controller into this new reference
            pController->CloneAnimationController(    nMaxOutputs,
                                                      nMaxSets,
                                                      nMaxTracks,
                                                      nMaxEvents,
                                                      &pReferenceController );

        } // End if we found an original object reference.

    } // End if Actor already exists
```

We have now seen the code block that is executed if an actor already exists that shares the same X file as the reference entity we are currently processing. If this code has been executed, the local pReferenceController variable stores a pointer to the interface of a cloned animation controller that will be assigned to the new CObject we are about to create. The local variable pReferenceActor also points to the existing actor that we intend to assign to the new CObject.

The next section of code is executed if the X file whose filename is stored in the reference has not yet been loaded into an actor. When this is the case, we create a new actor just like we always have and use the CActor::LoadActorFromX function to load the X file into the actor. We also register the CScene attribute callback function since our application always uses actors in managed mode.

```
    else
    {
        // Allocate a new Actor for this reference
        CActor * pNewActor = new CActor;
        if (!pNewActor) return false;

        // Load in the externally referenced X File
        pNewActor->RegisterCallback( CActor::CALLBACK_ATTRIBUTEID,
                                     CollectAttributeID,
                                     this );

        hRet = pNewActor->LoadActorFromX( Buffer, D3DXMESH_MANAGED,
                                          m_pD3DDevice );
        if ( FAILED(hRet) ) { delete pNewActor; return false; }

        // Store this new Actor
        if ( AddActor( ) < 0 ) { delete pNewActor; return false; }
        m_pActor[ m_nActorCount - 1 ] = pNewActor;

        // Store as object reference Actor
        pReferenceActor = pNewActor;

    } // End if Actor doesn't exist.
```

At this point a new actor has been added to the scene's CActor array and the actor will have been fully populated with the data from the X file. If the X file contained animation data, the actor will currently store a pointer to a controller. That is where it will remain until a second CObject is created that also references it (as shown in the previous code). Finally, at the bottom of this code block we assign the local pReferenceActor pointer to point to this actor. By doing this we know that whichever of the two code blocks were executed above, the pReferenceActor variable will point to the actor that needs to be assigned to the new CObject we are about to create.

All that is left to do is allocate a new CObject structure and add it to the scene's CObject array before assigning the actor and controller pointers to that object. We also store the world matrix of the reference entity that was passed into the function.

```
    // Now build an object for this Actor (standard identity)
    CObject * pNewObject = new CObject( pReferenceActor );
    if ( !pNewObject ) return false;

    // Copy over the specified matrix and store the colldet object set index
    pNewObject->m_mtxWorld        = mtxWorld;

    // Store the reference animation controller and action status (if any)
    pNewObject->m_pAnimController = pReferenceController;


    // Store this object
    if ( AddObject() < 0 ) { delete pNewObject; return false; }
    m_pObject[ m_nObjectCount - 1 ] = pNewObject;

    // Success!!
    return true;
}
```

One important point to make about the above code is that if the second code block is executed (i.e., the actor did not exist prior to the function being called) then the pReferenceController pointer will be set to NULL and we will be assigning NULL to the object's controller pointer also. This is because, at this point only one CObject references that actor and the actor currently manages the only animation controller. This relationship will change if the function is called again later to process another reference to the same X file. In that case, the first code block is executed and the controller is removed from the actor and stored in the object that was created in the prior call. The controller is then cloned and copied into the new CObject that is created at the tail of the function.

For completeness we will also show the CScene::LoadSceneFromX function. It has been updated with an additional line that registers a callback key handler with the actor that is created.

CScene::LoadSceneFromX

Students already familiar with our application framework will know that the CScene::LoadSceneFromX method is called by CGameApp::BuildObjects when the application first starts. It is passed the filename of the X file to be loaded. It is this loading function that is used by Lab Project 10.1 to load a single X

file that contains our small space scene. We will quickly look at this function code since it demonstrates an important topic that was discussed in this workbook. It calls the actor's RegisterCallback function to register a callback key registration function with the actor. When this function is used to load a scene (instead of LoadSceneFromIWF) the scene will only contain a single actor. As only one actor exists, the actor will always own the controller and the CObject that references it will always have its controller pointer set to NULL. This loading function is much simpler since it does not have to perform the controller attach/detach process.

This function is virtually identical to the implementation we looked at in the last workbook. It creates a new CActor and uses the CActor::LoadActorFromX method to load the requested X file. It then adds this actor's pointer to the scene's CActor array. It also allocates a new CObject and attaches the actor to it before adding this new object to the scene's CObject array. Remember, our scene class deals with CObjects at the top level so that it can handle the storage of both mesh hierarchies and single mesh objects in the scene. The scene's CActor array is not rendered from directly; it is where we store pointers to all the actors so we can release them in the scene's destructor.

Finally, this function sets up a few default lights so that we can see what we are rendering. Although the entire function is shown below as a reminder, notice that adding support for animated CActors has required only one new line of code (and even this line is optional). This new code is highlighted in bold in the following listing. It shows how we register the scene's callback key callback function with the actor. The actor will add this function's pointer, along with the passed context data pointer (the *this* pointer), to the actor's CALLBACK_FUNC array. As we saw earlier, when the actor is loading the X file, it will call this callback function and allow the function to add callback keys to any of its registered animation sets. As with the registration of attribute and texture callback functions discussed in the previous lesson, the callback key callback function must be registered before the call to CActor::LoadActorFromX is made.

```
bool CScene::LoadSceneFromX( TCHAR * strFileName )
{
    HRESULT hRet;

    // Validate Parameters
    if (!strFileName) return false;

    // Retrieve the data path
    if ( m_strDataPath ) free( m_strDataPath );
    m_strDataPath = _tcsdup( strFileName );

    // Strip off the filename
    TCHAR * LastSlash = _tcsrchr( m_strDataPath, _T('\\') );
    if (!LastSlash) LastSlash = _tcsrchr( m_strDataPath, _T('/') );
    if (LastSlash) LastSlash[1] = _T('\0'); else m_strDataPath[0] = _T('\0');

    CActor * pNewActor = new CActor;
    if (!pNewActor) return false;

    // Load in the specified X file
    pNewActor->RegisterCallback(CActor::CALLBACK_ATTRIBUTEID,CollectAttributeID,this);
    pNewActor->RegisterCallback(CActor::CALLBACK_CALLBACKKEYS,CollectCallbacks,this );
    pNewActor->LoadActorFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

```
    // Store this new actor
    if ( AddActor( ) < 0 ) { delete pNewActor; return false; }
    m_pActor[ m_nActorCount - 1 ] = pNewActor;

    // Now build an object for this mesh (standard identity)
    CObject * pNewObject = new CObject( pNewActor );
    if ( !pNewObject ) return false;

    // Store this object
    if ( AddObject() < 0 ) { delete pNewObject; return false; }
    m_pObject[ m_nObjectCount - 1 ] = pNewObject;

    // Set up an arbitrary set of directional lights
    ZeroMemory( m_pLightList, 4 * sizeof(D3DLIGHT9));
    m_pLightList[0].Type = D3DLIGHT_DIRECTIONAL;
    m_pLightList[0].Diffuse = D3DXCOLOR( 1.0f, 0.92f, 0.82f, 0.0f );
    m_pLightList[0].Specular = D3DXCOLOR( 1.0f, 0.92f, 0.82f, 0.0f );
    m_pLightList[0].Direction = D3DXVECTOR3( 0.819f, -0.573f, 0.0f );

    m_pLightList[1].Type = D3DLIGHT_DIRECTIONAL;
    m_pLightList[1].Diffuse = D3DXCOLOR( 0.4f, 0.4f, 0.4f, 0.0f );
    m_pLightList[1].Specular = D3DXCOLOR( 0.6f, 0.6f, 0.6f, 0.0f );
    m_pLightList[1].Direction = D3DXVECTOR3( -0.819f, -0.573f, -0.0f );

    m_pLightList[2].Type = D3DLIGHT_DIRECTIONAL;
    m_pLightList[2].Diffuse = D3DXCOLOR( 0.8f, 0.8f, 0.8f, 0.0f );
    m_pLightList[2].Specular = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 0.0f );
    m_pLightList[2].Direction = D3DXVECTOR3( 0.0f, 0.707107f, -0.707107f );

    m_pLightList[3].Type = D3DLIGHT_DIRECTIONAL;
    m_pLightList[3].Diffuse = D3DXCOLOR( 0.6f, 0.6f, 0.6f, 0.0f );
    m_pLightList[3].Specular = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 0.0f );
    m_pLightList[3].Direction = D3DXVECTOR3( 0.0f, 0.707107f, 0.707107f );

    // We're now using 4 lights
    m_nLightCount = 4;

    // Success!
    return true;
}
```

CActor - Conclusion

We have now covered all of the new code that has been added to the CActor class since Lab Project 9.1. While we have certainly added quite a large amount of code, much of it took the form of simple wrapper functions. Our actor now has built-in support for loading multiple mesh hierarchies, and is fully capable of animating of those hierarchies. Despite the myriad of code we studied here in this workbook, the API for the CActor class makes it extremely easy to use. Indeed, it can be initialized with only a few function calls by the application. This makes the CActor class an excellent tool for us as we move forward with our studies and can prove to be a powerful re-usable component in your own game projects. (And it will become even more potent when we add skinning support in the next chapter!)

Lab Project 10.2: The Animation Splitter

Our coverage of Lab Project 10.2 will be different from that of previous lab projects. We will not be covering the code in depth, but will instead discuss implementation at a high level. The reason we have decided to take this approach is that the application is essentially nothing more than a dialog box that calls the CActor functions we have already covered in this workbook. Therefore, you should have no trouble understanding the code should you decide to examine the source projects.

Design

A screenshot from the Animation Splitter is shown in Figure 10.3. The application has two main components. The first is the main render window which all of our applications have used. Just like our previous applications, this window has its contents redrawn each iteration of the game loop in the CGameApp::FrameAdvance method. Unlike our other lab projects however, there is no CScene class or CTerrain class for managing the objects in the world. This is primarily because we only load one X file at a time. The CGameApp object will load a single actor from an X file (inside CGameApp::BuildObjects) and will call CActor::DrawActor in the CGameApp::FrameAdvance method during each iteration of the main game loop. That is about the extent of the scene management that has to be done in this application.

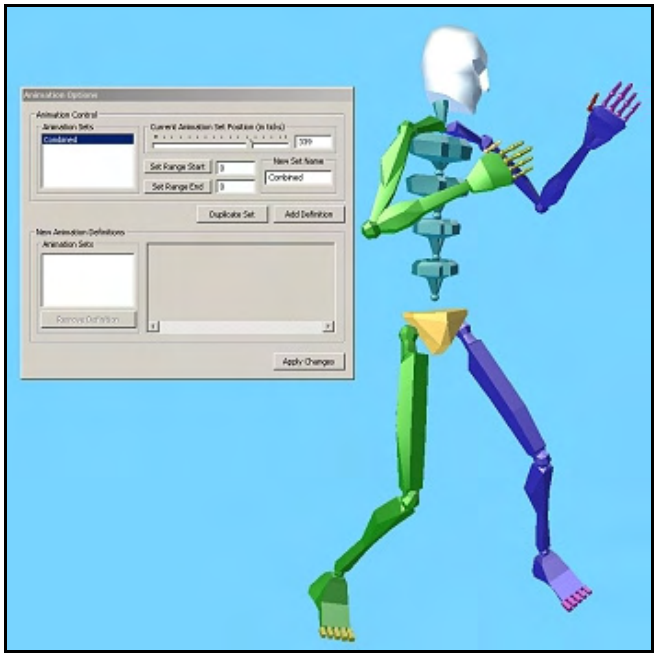


Figure 10.3: The Animation Splitter

In Figure 10.3, we see the main render window with our actor being rendered. In this example, we have loaded a file called ‘Boney.x’ which contains the skeleton of a humanoid character.

The main part of this application that is quite different from previous lab projects is the modeless dialog box that we create to provide a number of controls. We need the dialog because the purpose of this application is to provide a means for editing the animation data stored in the actor. More specifically, the purpose of this tool is to provide a means for us to take an actor's animation set and split it into multiple animation sets. Therefore, we have controls that allow us to navigate to a certain position on an animaton set's timeline and set start and end markers. We can then give this range a name and add it as a new split definition. So the controls on the dialog will not only display the contents of the animation data in the loaded X file, but they will also allow you to alter that data before re-saving the modified X file. Once we have added all the new definitions we wish to add, we can hit the Apply button. At this point, the new animation sets will be created from each split definition and the old animation sets will be released from the actor's controller. It should be obvious that this entire splitting process can be done by the application with a single function call to CActor::ApplySplitDefinitions. In order to understand how the splitter works internally, we shall first examine how it is used. Therefore, this section will also serve as a user handbook for the Animation Splitter.

Using the Animation Splitter

To demonstrate using the animation splitter, we will assume you are working with the X file 'Boney.x'. This X file contains a single animation set (called 'Combined') which contains a number of animation sequences combined into a single set. This is a typical scenario when exporting X files from applications that support only a single animation set and is the reason that this tool was created. It should be noted however, that the application works perfectly fine even if the X file already contains multiple sets. These multiple sets can also be subdivided into more sets if desired.

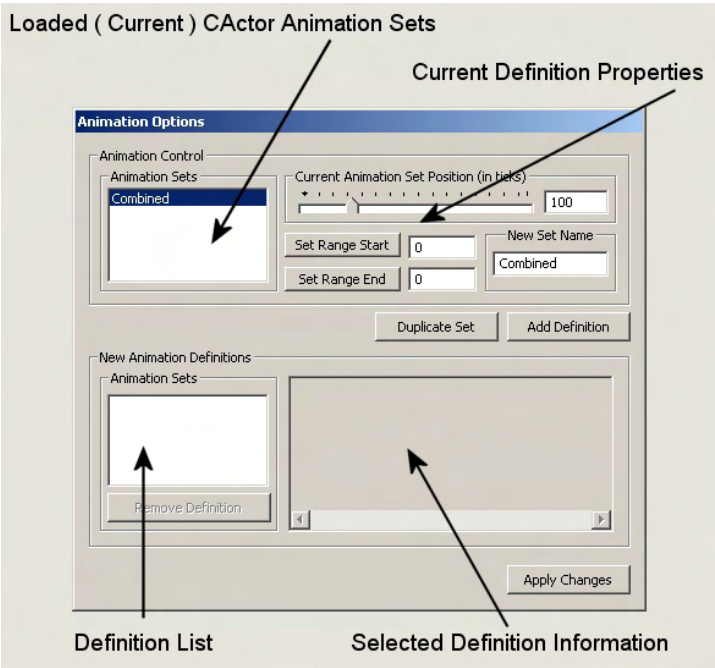


Figure 10.4: The Interface Components

In Figure 10.4 we see how the dialog controls might look after the ‘Boney.x’ file has been loaded. The uppermost list box contains animation sets that were loaded from the X file and are currently registered with the actor. As you can see, in our example the X file contained only one animation set called ‘Combined’. In Figure 10.4 we have selected this animation set, which enables the Split Definition control panel to the right of this list box (the slider, set range buttons, etc.).

Because we have selected the ‘Combined’ animation set in the list box, this tells the application that the next split definition we add will contain a subset of the animation data in this animation set. Remembering our discussion of CActor::ApplySplitDefinitions, this is the source animation set for the next split definition we are about to construct.

This application allows you to add split definitions to a list one at a time. The bottommost list box on the dialog shows the split definitions you have already added. Obviously, in Figure 10.4 this box is empty as we have not yet added any.

When a source animation set is selected from the top list box, the definition controls become active. We can move the slider to navigate to positions on the animation set’s timeline. The 3D render window will reflect the changes in the actor as we drag the slider through the actor’s timeline. This makes it easy to visually navigate to the correct position in the animation. The first thing we will want to do is set the start and end markers for this definition. In this example, we will extract an imaginary walking sequence that is positioned between ticks 100 and 200 on the timeline.

First we would move the slider to the start position and press the Set Range Start button. This will record the current position of the slider as the start marker for the definition we are currently building. We would then move the slider position again to where the walking sequence ends (200 in this example) and press the Set Range End button to record this position as the end marker. Figure 10.5 shows the dialog after we have recorded the start and end markers. Notice that the subsection of the timeline is also highlighted in blue on the slider.

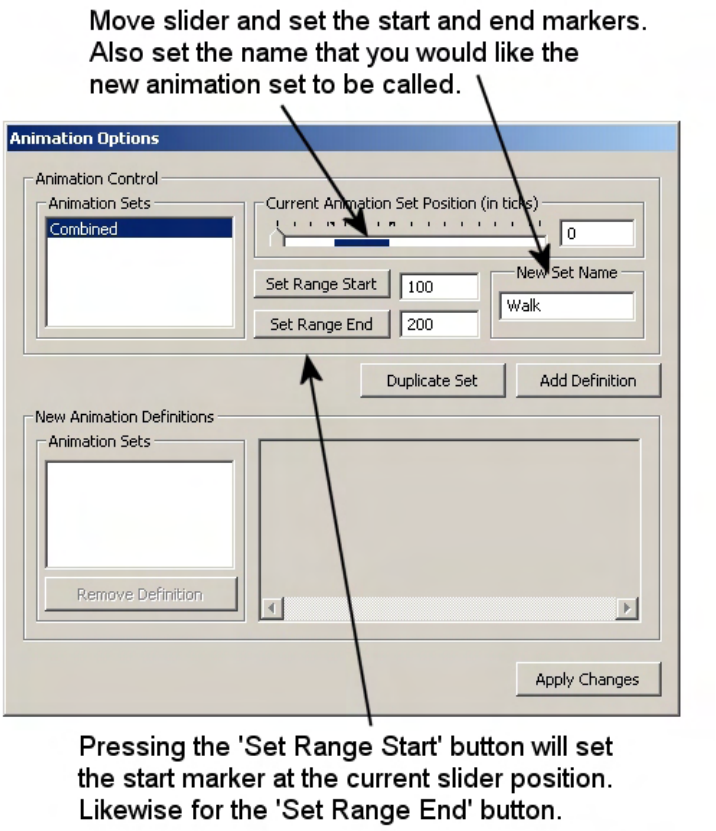


Figure 10.5: Setting up a new Split Definition

We have now marked the exact section of the original animation set that we would like to extract into a new one. That is, we are stating that we will want a new animation set to be built that will only contain the animation data that fell between 100 and 200 ticks on the timeline of the original animation set. We will also want to give this new animation set a name. As you can see in Figure 10.5, we assign it the name ‘Walk’. We just type the name into the text edit box to the right of the Set Range Start and Set Range End buttons.

At this point we have correctly set up the properties for our first split definition. We can now inform the dialog that we wish to add this split definition to a list and move on to defining another one. We add the properties to the split definitions list by pressing the Add Definition button.

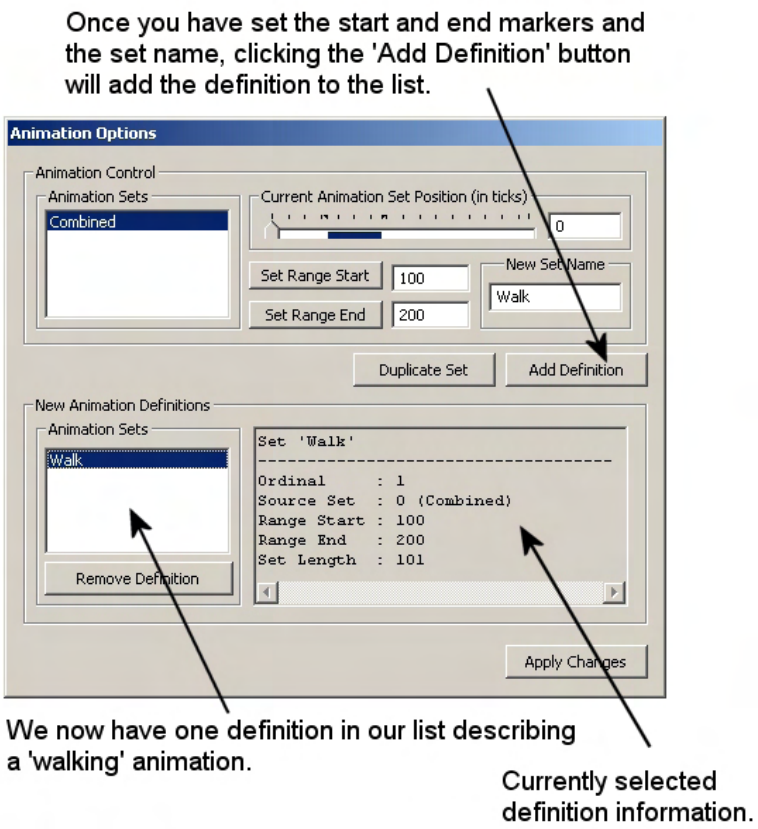


Figure 10.6: Adding the Split Definition

Once the new definition has been added you will see it appear in the bottom list box. As Figure 10.6 shows, this box contains the name of our newly added ‘Walk’ animation. What is important to realize is that at this point, no animation sets have been physically created. All we are doing is adding split definition structures to an array one at a time. This allows the original source animation sets to remain in the controller so that they can be sources for subsequent definitions we may wish to create.

At this point we have added one split definition to our list. Continuing this example we will now definine a second animation set which we will call ‘Run’. For the sake of this explanation, we will imagine that the keyframes for this sequence are also contained in the original ‘Combined’ animation set between tick positions 300 and 400.

With our new ‘Walk’ definition added to the list, we can now set the definition controls for the next definition we wish to add. In Figure 10.7 you can see that the slider has been used to set two new start and end marker positions on the animation timeline. This time, the markers bound a 100 tick range between positions 300 and 400. We then type the name (‘Run’) we would like this animation set to be called in the New Set Name edit box. We have now filled out all the information for our second split definition.

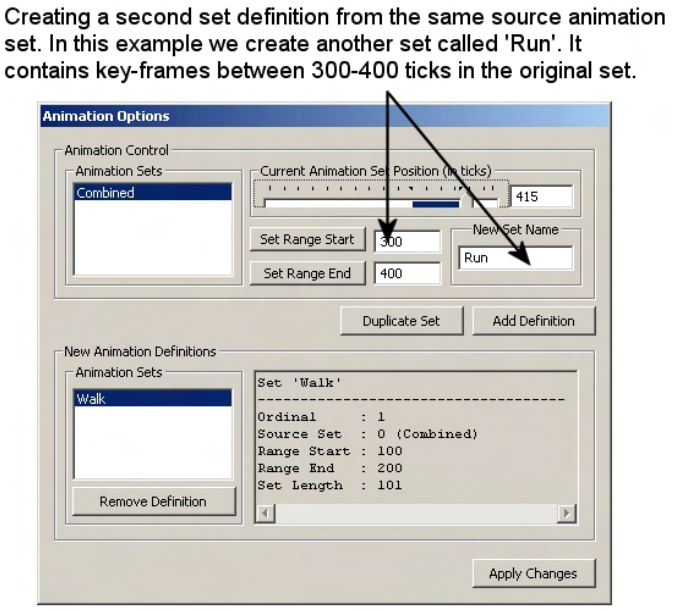


Figure 10.7: Setting up a second Split Definition

With the definition controls now containing the information for our second definition, we add it to the split definition list. We once again press the Add Definition button which creates a new split definition and adds it to the current array. If we look at the bottom list box in Figure 10.8, we can see that after this step is performed there are now two definitions in our list: ‘Walk’ and ‘Run’.

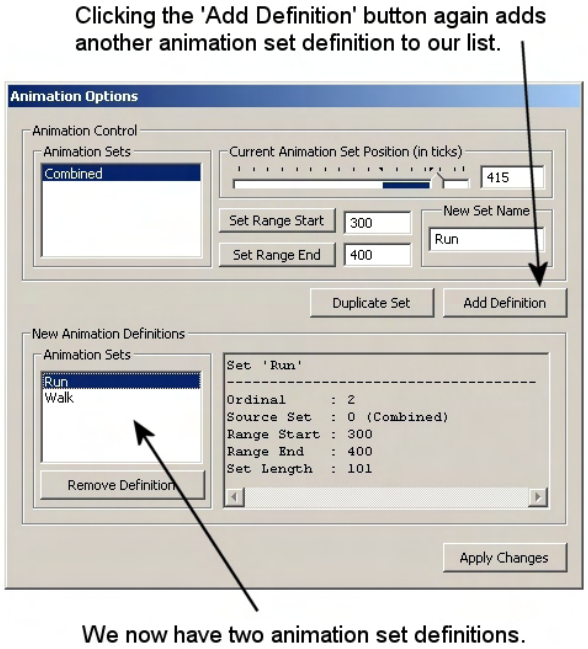


Figure 10.8: Adding the Second Definition

Assuming that we only wish to break the source animation set into two sets in this example, our work is done. It is now time to instruct the application (more specifically, the actor) to apply these split definitions.

When the Apply Changes button is pressed, the current array of split definitions stored in the dialog object is passed to the CActor::ApplySplitDefinitions method. We know that this method will create a new animation set from each definition and register them with the actor's animation controller. At this point, the original animation set(s) will cease to exist. The actor has now been modified in memory and its original animation set(s) have been replaced by the the new animation sets described by the definitions.

Finally, Figure 10.9 shows how the dialog would look after this step. At this point, the definitions list is empty since there are no longer any pending split definitions to process. Those definitions have now been turned into real animation sets and therefore, they are now listed in the top list box instead of the original source set. At this point, you could perform more subdivision steps on these new animation sets, although this is something you will probably not need to do very often.

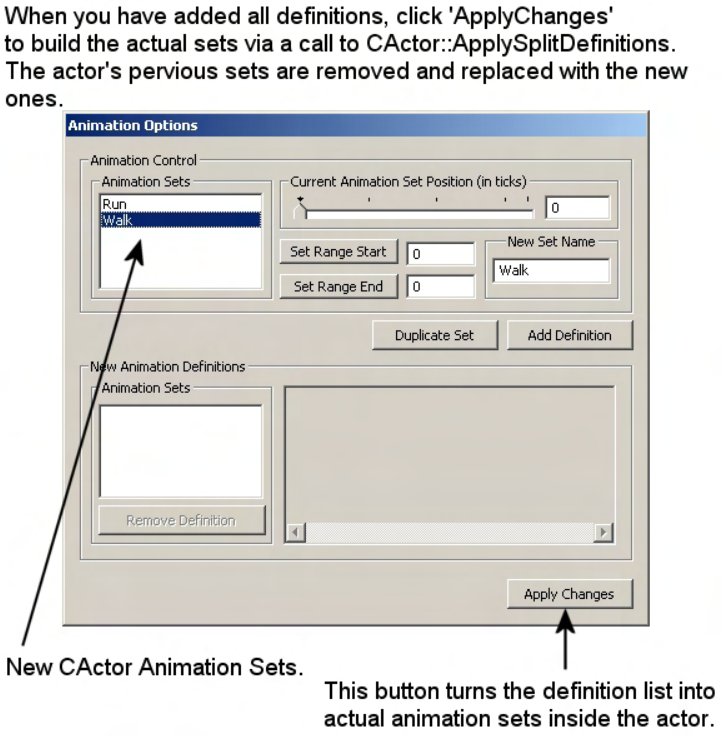


Figure 10.9: Building Animation Sets from Split Definitions

The actor's animation sets have now been changed, but no editing of the original X file has been done. Usually, when splitting animations in this way you will want to save the modified actor back out to the X file. You can do this by selecting the Save option from the file menu in the main render window.

Implementation

Most of the code to this lab project has already been covered, due to the fact that the majority of its functionaility is within CActor. We are also quite familiar with the CGameApp class, having used it in all of our lab projects from the beginning of this course series. Although we are going to make one or two changes to this class, for the most part, it is the same in this project as all the other versions that have come before it. One of the important new changes in CGameApp happens in the CGameApp::CreateDisplay method. This method has been used right from Lesson One to build our enumeration object and create the main render window of the application. Now, at the bottom of that function we also make the following call:

```
m_AnimDialog.InitDialog( m_hWnd, &m_Actor )
```

m_AnimDialog is a variable of type CAnimDialog, which is a class we will write in this lab project. This dialog class is responsible for tracking changes to controls and reflecting those changes back to the actor. The CAnimDialog::InitDialog method is the method that actually instructs the CAnimDialog object to display the modeless dialog on the screen. The dialog box layout is defined as a resource, so controls are automatically created for us when the call to the Win32 function CreateDialogBoxParam is made.

The InitDialog method accepts two parameters that our dialog class will need. The first parameter passed in by CGameApp is the handle of the main render window. Although the dialog will be modeless and not confined to being positioned within the bounds of the render window, the render window will still be its parent and the window that contains the application menu. When the parent window is destroyed, so is the child dialog.

The second paramater passed by CGameApp is the address of the actor that has been loaded. The dialog class will need access to the actor so that it can call its ApplySplitDefinitions function when the Apply button is pressed. It will also need to alter the position of the currently selected animation set in response to the user dragging the slider bar. This will ensure that when the slider bar is moved to a new position, the actor's hierarchy is updated to reflect that position change. Since the CGameApp object is consistently rendering the actor in a loop (via CGameApp::FrameAdvance), those changes made to the actor's hierarchy will be immediately reflected in the render window when the actor is redrawn in the next iteration of the render loop.

> **Note**: We are creating a modeless dialog box so that the program will not have to wait for the user to finish applying definitions before our CGameApp code can continue execution. As soon as InitDialog has created the dialog box, program flow will return back to CGameApp and the main render loop will be entered. The dialox box will remain on the screen processing input and relaying these property changes to the actor.

CAnimDialog – A Brief Overview

We will not step through the code to the CAnimDialog class line by line as we have done in other lab projects. It uses the most basic Win32 concepts (dialogs and controls) and most of the code is just concerned with setting and getting the properties of these controls and handling any messages they send and receive. Win32 controls and dialog box programming is discussed in detail in **C++ Programming for Game Developers: Module II** here at the Game Institute, so we will assume that you have either taken this course already, or that you are familiar with Win32 user interface programming techniques. We will not explain the concepts of Win32 dialog box and control programming here since it falls outside the scope of what we intend to teach in this course. If you are not comfortable with this type of Win32 programming, then we strongly recommend you take the abovementioned course before continuing (or in conjunction with) with your Graphics Programming studies.

Below, we see the CAnimDialog class declaration from CAnimDialog.h. The methods of the class are not shown here, but there are quite a few. Most of these methods are functions that react to control changes while others are called to set the default values for the controls. What we do wish to focus on here are the member variables of this class and how they are used by this object.

```
class CAnimDialog
{

 // Methods not shown here

 private:
    //-----------------------------------------------------------------------
    // Private Variables for This Class
    //-----------------------------------------------------------------------
    HWND                        m_hDialog;
    CActor                    * m_pActor;
    LPD3DXKEYFRAMEDANIMATIONSET m_pCurrentAnimSet;
    ULONG                       m_nCurrentAnimSet;
    ULONG                       m_nStartTick;
    ULONG                       m_nEndTick;

    ULONG                       m_nDefCount;
    AnimSplitDefinition         m_pDefList[MAX_DEFINITIONS];

    HFONT                       m_hFontFW;

};
```

HWND m_hDialog

When the InitDialog method instructs Windows to display the dialog, we are returned a handle to the dialog window. This handle will be needed to send and receive messages to the dialog and its controls. For example, when we wish to send a message to the dialog window (such as to enable the window on screen), we will need to send a message using the SendMessage function. This Win32 function expects the first parameter to be the handle of the window it is expected to send the message to.

CActor * m_pActor
This member is used to cache a pointer to the actor that the dialog controls will manipulate. This pointer is passed in by the CGameApp object when it issues the call to CAnimDialog::InitDialog. This provides the dialog object with a pointer to the same actor the CGameApp object is rendering. The dialog will need access to the actor on several occasions. For example, when the user alters the slider position, the dialog object will need to retrieve the new slider position and use it to set the position of the actor's currently active animation track. The dialog object will also need to communicate with the actor when the user selects a new source animation set. When a new source animation set is selected, the dialog object will retrieve the index of the animation set from the top list box and tell the actor to assign this animation set to track 0. Since we can only be editing using one source animation set at a time, the dialog can always make sure that that the currently selected animation set is bound to track 0 on the actor's mixer. That way, it can safely assume that when the slider position is changed by the user, it has to apply this position change to track 0.

LPD3DXKEYFRAMEDANIMATIONSET m_pCurrentAnimSet
This member is used to store a pointer to the interface of the currently selected source animation set. When the user selects a source animation set from the top list box, it becomes the current animation set. Changing the position slider will allow us to step through the animations in this set. Therefore, when a new set is selected, the dialog object will fetch the animation set interface from the actor and store it here. Now it can assign it to track 0 on the mixer and extract other information from it as needed to populate the controls. For example, the slider bar must represent the timeline of the animation set, so it will be necessary to extract the period of the animation set, convert it to ticks and store this range in the slider control.

ULONG m_nCurrentAnimSet
This member is used alongside the previous member. Where as the previous member stores the interface pointer of the currently selected source animation set, this member stores the index of this animation set assigned to it by the animation controller. This allows us to reference this animation set in the actor by index.

ULONG m_nStartTick
This member will be used to store the current value for the Set Range Start edit box. This is the currently set start marker position used for marking a range on the current animation set's timeline. This value will be used as the start marker when the definition is added to the list.

ULONG m_nEndTick
This member will be used to store the current value for the Set Range End edit box. This is the currently set end marker position used for marking a range on the current animation set's timeline. This value will be used as the end marker when the definition is added to the list.

ULONG m_nDefCount
This member contains the number of split definitions that have been added to the dialog's split definitions array. This will be zero when the application starts.

AnimSplitDefinition m_pDefList[MAX_DEFINITIONS]
This is an array of AnimSplitDefinition structures. Each element in this array will have been added in response to the Add Definition button being pressed on the dialog. When that happens, the split properties (end marker position, start marker position, and set name) are copied from the dialog controls into a new AnimSplitDefinition structure in this array. The definition count is then incremented.

HFONT m_hFontFW;
This is a handle to the font we use in the dialog.

While we will not look at all the helper functions for this class, we will look at a few. The first function we will look at is the dialog box procedure. This is the method that is sent the messages that are generated by the controls (sent via the operating system).

CAnimDialog::OptionsDlgProc

This function handles the basic message processing for the dialog. It is called by Windows whenever a message is generated for our dialog box. For example, if the user pressed the Apply button, a WM_COMMAND message will be sent to this function. The message will also be accompanied by the ID of the control that sent it. For example, if a WM_COMMAND message has been triggered by a control with the id IDC_APPLY, we know the user has pressed our Apply button because IDC_APPLY is the id we assigned to that button in the resource editor.

Although this function is simple (it is just a switch statement that calls processing functions) we will discuss the messages it recieves and the helper functions it calls as a result. We will not be covering those helper functions in this workbook since their code is very straightforward. You should have no problem looking through the source code since it is just basic Win32 controls programming. This function will show you how and why those helper functions get called. We will also explain here what those helper functions actually do. This will give you a better overview of the entire application without us having to examine code.

The CAnimDialog::OptionDlgProc function is called by the operating system (indirectly via a static routing function) whenever a message is generated for the dialog window or one of its controls. We can simply process the messages we want and return false for any that we do not. We return false to indicate that we did not handle the message and that Windows should still perform default processing on it. Luckily, the OS implements a lot of default behavior so that the dialog window knows when to paint itself (for example) or when it is overlapped by other windows. Therefore, we only have to intercept messages which we intend to process using our own additional logic.

When this function is called it is passed the handle of the window that generated the message, the message itself (Windows defines a large set of message codes as UINTs) and two DWORD parameters (wParam and lParam). The contents of the final two parameters will depend on the message being received, as we shall see.

The first section of the code sets up a switch statement to test the Message parameter for messages we are interested in.

```
BOOL CAnimDialog::OptionsDlgProc( HWND hWnd, UINT Message,
                                  WPARAM wParam, LPARAM lParam)
{
    ULONG  i;

    switch ( Message )
    {
    case WM_INITDIALOG:

        // Store dialog here since this is fired before the
        // CreateDialog has returned
        m_hDialog = hWnd;

        // Populate the dialog
        PopulateOptions();

        // Set focus for us.
        return TRUE;
```

The first case happens when the dialog window has received the WM_INITDIALOG message. The operating system will send this message to all dialog box windows just after the window has been created but before it has been displayed. This allows you to perform some default initialization such as filling up the list box controls with the information you want in them. As you can see, we do exactly that.

First we store the handle of the dialog window in the CAnimDialog's member variable, so this C++ object always has a handle to the GUI element to which it is attached (i.e., the dialog window itself). This is the first time we get to know about the handle of the window because the operating system has just created it for us. We store it because we will need it later when sending messages to our controls. Remember, at this point, Windows has created our dialog box and all the control windows that were specified in the resource template. However, these controls and the dialog window are not visible to the user at the moment. All controls (list boxes for example) will also be empty of content at this point.

After we have stored the window handle, we call the PopulateOptions method. This method does quite a bit of preparation. It first loops through every animation set in the actor and extracts its name and index. Each name is added to the top list box and the index of the animation set in the animation controller is assigned to the list box element as item data. When the user selects an item in this list box, we can extract the name they selected and the index of the animation set they would like to use as the new source set. This method will also initially disable the slider control because no source animation set will be selected initially. The slider is only used to scroll through the position of the currently selected source animation set and serves no purpose if a source animation set has not yet been selected.

When the PopulateOptions method returns, the top list box will be filled with all the animation set names currently managed by the actor. With the dialog controls now initialized, processing of this message is complete. We return true to let the OS know that we have handled this message and require no further default processing of this message.

In the next case, we test to see if the message passed is a WM_COMAND message. Many controls send WM_COMMAND messages. When a WM_COMMAND message has been sent to us by Windows, the

wParam parameter will contain vital information. In the low word of this variable will be stored the numerical ID assigned to the child control of the dialog (in the resource editor). This is the control that is sending the message and it usually means that the user has interacted or changed the contents of the control in some way. In the high word of wParam will be the actual event (notification message) that has occurred.

The following sections of code are all for the WM_COMMAND message case. In this first code block, we can see that if the WM_COMMAND has been received, we enter another switch statement to test which control sent the message. Since the control ID is stored in the low word of the lParam parameter, we can use the LOWORD macro to crack it out and examine it.

```
    case WM_COMMAND:

        switch ( LOWORD(wParam) )
        {
```

Now that we know we have a WM_COMMAND message, let us see what control sent it. The first test is for the control with the ID of IDC_LSTANIMSETS. This is the ID that was assigned to the top list box on the dialog box in the resource editor. It is the list box that will, at this point, contain the names of all animation sets currently registered with the animation controller. If the contents of LOWORD(wParam) matches this ID, then the user has performed some action in this list box which has triggered this message. For example, the user may have selected another animation set from the list. When this has happened, the list box control will always send a notification message to the parent window to notify it that a selection change has occurred.

Once we know it is the list box control that sent the message, we can test the high word of the wParam parameter for the notification code that was sent. The only event/notification we are interested in intercepting is the LBN_SELCHANGE list box notification message. Windows will send us this message when the user has changed the selection. In our example, it means the user has changed their choice of source animation set. You can see in the next section of code that if this is our list box that has sent the message, and if the notification it has sent is one of a selection change, we will call the CAnimDialog::CurrentSetChanged method. This method will extract the new selected animation set name and index from the list box. It will then use that set index to fetch the matching animation set interface from the actor and assign it to track 0 on the mixer. The track will be assigned an initial position of 0. This new animation set will now be the current animation set that will be manipulated by the slider control and will be the new source animation set for the split definition currently being constructed. Notice how we return true once we have processed the message so that Windows knows we have handled it.

```
        case IDC_LSTANIMSETS:
            if( LBN_SELCHANGE == HIWORD(wParam) )
             { CurrentSetChanged(); return TRUE; }
            break;
```

If the WM_COMMAND message was sent by a selection change in the bottom list box then the control sender ID will be IDC_LSTNEWANIMSETS. This is the list box which displays the currently compiled split definitions. If the user has selected one of these sets from the list, then as in the above case, a

selection change notification will be sent by that list box control. When a user selects an already added split definition from the definitions list, the details of the split definitions are displayed in the status window next to the list box. That is essentially the task of the CAnimDialog::NewSetChanged method. This helper function extracts the values from the selected definition structure and displays them in the status window.

```
        case IDC_LSTNEWANIMSETS:
            if( LBN_SELCHANGE==HIWORD(wParam) ) { NewSetChanged(); return TRUE; }
            break;
```

If the WM_COMMAND message sent was generated by the user pressing the Set Range Start or Set Range End buttons, then the IDs of the control sending the message will be IDC_BUTTON_SETSTART or IDC_BUTTON_SETEND, respectively. Once again, these are the IDs we assigned to these button controls in the resource editor. When this is the case, we call the SetRangeStart and SetRangeEnd methods. These methods simply extract the current position value from the slider control and store it in the m_nStartTick and m_nEndTick member variables depending on which of the two buttons was pressed. These values are cached in these member variables until the Add Definition button is pressed. At that time, the start and end marker values are copied into a new split definition structure in the dialog's definition array.

```
        case IDC_BUTTON_SETSTART:
            SetRangeStart();
            break;

        case IDC_BUTTON_SETEND:
            SetRangeEnd();
            break;
```

When the user has set the start and end markers for the current definition and wants to add that definition to the list, the Add Definition key is pressed. This button (whose ID is IDC_ADD_DEFINITION) sends a WM_COMMAND message. It is trapped in the next section of code and used to trigger the AddSetDefinition method. This method adds a new split definition to the end of the array. The previous cached start and end marker positions are copied into this structure and the method extracts the text the user has typed into the Set Name edit box and stores it in the split definition structure as the new animation set's name. There is also a Remove Definition button which, when pressed, triggers a call to the RemoveDefinition function. You can probably guess what this function does.

```
        case IDC_ADD_DEFINITION:
            AddSetDefinition();
            break;

        case IDC_DEL_DEFINITION:
            RemoveDefinition();
            break;
```

At some point, the user will have added all the desired split definitions to the dialog object's internal array of definition structures. At such a time, they will click the ApplyChanges button and trigger a WM_COMMAND message with the control ID of IDC_APPLY. As you can see in the following code

block, this will trigger a call to the CAnimDialog::ApplyChanges method. This method simply calls the CActor::ApplySplitDefinitions method and passes in the array of split definitions that has been compiled. When this function returns, the new animation sets will have been built and registered with the actor's animation controller and the old animation sets (the original source animation sets) will have been deleted. The following code shows the last sub-case for the WM_COMMAND message:

```
        case IDC_APPLY:
            ApplyChanges();
            break;

        } // End Control Switch
        break;
```

Not all controls send their notification messages on the back of WM_COMMAND messages. One such control is the slider control, which is not one of the original Windows controls. Instead, it belongs to a pool of controls referred to as the *common controls*. These controls were added after the core controls as a means of extending the user interface capabilities of the OS.

When a slider control is moved or changed it sends a WM_HSCROLL message (instead of a WM_COMMAND message) to the parent window. The lParam parameter will contain the *handle* of the control window that sent the message (i.e., the slider). However, what we actually want is the window ID (not its handle) just as had in the above cases, so that we can make sure that the ID of the control that sent this scroll message is indeed the ID we assigned to our animation set slider control in the resource editor.

Win32 has a function called GetDlgCtrlID which accepts a window handle and returns its control ID. In this next section of code, we are interested in scroll activity in our slider bar which has a control ID of IDC_ANIMSLIDER (assigned in the resource editor). When this is intercepted, the CAnimDialog::AnimSliderChanged method is called, which extracts the current position of the slider and uses it to update the position of track 0 on the animation controller. This is the track that the current source animation set is assigned to. This function will also force the actor to perform a hierarchy update pass so that when the CGameApp::FrameAdvance method renders the actor in the next iteration of the render loop, these changes are reflected in the output. We then return true from this function so that Windows knows we have handled this message.

```
    case WM_HSCROLL:

        switch ( GetDlgCtrlID( (HWND)lParam ) )
        {
        case IDC_ANIMSLIDER:

            AnimSliderChanged();

            // We handled it
            return TRUE;

        } // End Control Switch
        break;
```

```
    } // End Message Switch

    // Not Processed!
    return FALSE;
}
```

If the end of this function is reached, then it means that a control that we are not interested in sent the message, or a control that we are interested in sent a message of some type that we are not setup to handle. When this is the case, we return false so that Windows knows that we took no action and it can feel free to perform any default processing it may want to do. For example, we would want Windows to handle paint messages for controls when their contents need to be refreshed.

CAnimDialog::AnimSliderChanged

If we look at the source code to the CAnimDialog::AnimSliderChange function (called from the dialog procedure in response to a WM_HSCROLL message), we can see the typical interaction between the dialog box object, its controls, and how changes to those controls update the properties of the actor. This will enlighten you with respect to how the messaging system works. Remember, if your Win32 is a little rusty, controls and dialog box programming are all covered in Module II of the C++ Programming for Game Developers series here at the Game Institute.

The function first checks that a source animation set has been selected. If this is the case then the interface for that animation set will have been cached in the m_pCurrentAnimSet member variable. This interface pointer would have been extracted from the actor and stored in this member variable in response to the user selecting a new source animation set from the list (see the CAnimDialog::CurrentSetChanged method). Provided this is the case, we then need to get the window handle of the slider control using the GetDlgItem function. This function accepts, as its first parameter, the handle of the dialog box, and as the second parameter, the ID of the control we wish to get an HWND for. The ID of our slider control is IDC_ANIMSLIDER.

```
void CAnimDialog::AnimSliderChanged()
{
    HWND   hControl;
    TCHAR  Buffer[128];
    double Position, MaxSeconds;
    ULONG  TickPos = 0;

    // Break out of here if we don't have one selected atm
    if ( m_pCurrentAnimSet )
    {
        // Get the slider control hwnd
        if ( !(hControl = GetDlgItem( m_hDialog, IDC_ANIMSLIDER )) ) return;
```

If we get this far, then we have an HWND to the slider control stored in the hControl local variable. We then get the period of the currently select source animation set so we can make sure that the position they have provided is not out of the range.

```
        // Store max seconds variable
        MaxSeconds = m_pCurrentAnimSet->GetPeriod();
```

Next we need to get the position of the slider. When the animation set was first selected by the user in the CurrentSetChanged method, the period of the animation set would have been retrieved, converted to ticks, and then used to set the range of the slider control. This means that the slider control's range should be between 0 and MaxTicks at its start and end positions. We now need to extract the current position that the user has moved the slider to. We do this by sending the TBM_GETPOS message to the slider control using the Win32 SendMessage function. The function will return the current position of the slider (in ticks) so that we can set the current position of the track that the animation set is assigned to.

```
        // Retrieve the new animation set position in ticks
        TickPos = ::SendMessage( hControl, TBM_GETPOS, (WPARAM)0, (LPARAM)0 );
```

When we set the track position of the actor, we specify the new position in seconds, not ticks. So we must convert the new tick position we have just extracted from the slider control into seconds by dividing the tick position by the animation set's ticks per second ratio. We do not want the position to ever be greater than the period of the animation set; otherwise the periodic position of the animation set will loop back around to zero. Therefore, we calculate the new position as being the minimum of either the new track position extracted from the slider (in seconds) and the period (MaxSeconds) of the animation set. We never want the position to be greater than the period, so this test is done for safety. Notice we subtract epsilon from MaxSeconds to account for floating point errors and to make sure that the position is always clamped between the start and end of the animation set's period.

```
        // Convert to seconds for track (clamp - epsilon to prevent rendering loop
        //  around)
        Position = min( (double)TickPos /
                        m_pCurrentAnimSet->GetSourceTicksPerSecond(),
                        MaxSeconds - 1e-4 );
```

We now have the new position for track (always track 0 in this application) as described by the slider control. We use the CActor::SetTrackPosition method to apply this position change to the track. We then update the actor's absolute matrices by calling the AdvanceTime method. Notice how we pass 0.0 as the first parameter because we do not want to increment any track positions since we have just set the track position to the correct value. However we do pass in 'true' as the second parameter, which we know from our ealier discussion forces the actor to traverse the hierarchy and build its absolute marices for each frame. The next time the CGameApp::FrameAdvance method renders the actor, its matrices will have been updated and the new position of the actor (described by the new slider position) will be reflected in the render window.

```
        // Update track time
        m_pActor->SetTrackPosition( 0, Position );

        // Ensure all frame matrices are up to date
        m_pActor->AdvanceTime( 0.0f, true );

    } // End if animation set
```

At this point the actor has been updated, but we also have an edit box to update. Next to the slider is an edit box which displays the current position of the slider numerically for greater accuracy during the setting of the start and end marker positions. As the slider has now been updated, we should also replace the text in this edit box with the new position as well.

We do this by using the GetDlgItem method to retrieve the HWND for the edit window. The ID of this edit window is defined in the dialog template as IDC_EDIT_CURRENTPOS. Once we have the window handle, we convert the tick position that we just extracted from the slider into a string stored in a local char array (Buffer). We then change the text of the edit window to the value stored in this string using the Edit_SetText macro (defined in windowsx.h).

```
    // Get the edit control
    if ( !(hControl = GetDlgItem( m_hDialog, IDC_EDIT_CURRENTPOS )) ) return;

    // Get a string of the position in ticks
    _itot( TickPos, Buffer, 10 );

    // Set the value
    Edit_SetText( hControl, Buffer );
}
```

Note: The Edit_SetText macro is just a friendly wrapper around the SendMessage function used to send messages to all windows. All the core controls (list boxes, edit boxes, buttons, etc.) have macros defined for them just like this one in windowsx.h. They make the setting and getting of control properties easier and more user-friendly. Notice however that we do not use a macro like this when getting the position of the slider control. This is because the slider control is not one of the original core controls of the operating system and therefore has none of these macros defined for it. We have to do it the slightly more involved way for the slider (and any common control) by making the call to the SendMessage function ourselves. All the Edit_SetText macro is doing is wrapping the sending of the WM_SETTEXT message to the edit control using the SendMessage function.

CAnimDialog::CurrentSetChanged

CurrentSetChanged is called from the dialog box procedure in response to the user selecting a new item in the top list box on the dialog. This means the user has selected a new source animation set that it wishes to split.

The first thing we do is retrieve the HWND of the list box that the animation set names (and indices) are stored in. We use the GetDlgItem function again to retrieve the HWND of the list box with the control ID of IDC_LSTANIMSETS. We then use the ListBox_GetCurSel macro (see windowsx.h) to retrieve the current item that is selected in the list box as an integer index. Remember, this function was called because the user selected a new item. It is the index of this newly selected item that is returned from this function.

```
void CAnimDialog::CurrentSetChanged()
{
    HWND                          hControl = NULL;
    LPD3DXANIMATIONSET            pAnimSet = NULL;
    LPD3DXKEYFRAMEDANIMATIONSET pKeyAnimSet = NULL;
```

```
    // Get the current selection
    if ( !(hControl = GetDlgItem( m_hDialog, IDC_LSTANIMSETS )) ) return;
    ULONG nIndex = ListBox_GetCurSel( hControl );
```

We now know the index of the item in the list box that the user has just selected and we can use it to fetch the corresponding animation set from the actor and cache it in the member variables of the CAnimDialog object. First, let us release any current animation set information that may be cached from a previous selection.

```
   // Release the previously selected animation set
    if ( m_pCurrentAnimSet ) m_pCurrentAnimSet->Release();
    m_pCurrentAnimSet = NULL;
    m_nCurrentAnimSet = 0;
```

Next we test that we did get a valid selection index from the list box. If not, then it means the user has deselected an item, leaving no item selected. When this is the case, we call the CAnimDialog::EnableSetControls method, which disables all the slider and marker controls. They have no meaning if a source animation set is not selected. Also, if no animation set is selected, we make sure we also remove any previous current animation set from the first track on the mixer.

```
    // No item / error ?
    if ( nIndex == LB_ERR )
    {
        // Disable the animation set controls
        EnableSetControls( false );

        // Actor uses no set atm.
        m_pActor->SetTrackAnimationSet( 0, NULL );

    } // End if nothing selected
```

If a valid index was returned, then it is time to find out which animation set this index maps to in the actor. We can then fetch the interface of this animation set from the actor and assign it to track 0 on the mixer. Notice how each string in the list box also has assigned (as user data) the index of the animation set in the controller that the item maps to. Therefore, we use the ListBox_GetItemData to retrieve the index of the set we need to fetch from the animation controller.

```
    else
    {
        // Retrieve the item data for this
        ULONG nAnimSet = ListBox_GetItemData( hControl, nIndex );

        // We're now using this animation set
        pAnimSet = m_pActor->GetAnimationSet( nAnimSet );

        // We can only split up keyframe animation sets. Ensure this is available
        if ( FAILED(pAnimSet->QueryInterface( IID_ID3DXKeyframedAnimationSet,
                                              (LPVOID*)&pKeyAnimSet )) )
        {
            // Disable the animation set controls
            EnableSetControls( false );
```

```
            // Actor uses no set.
            m_pActor->SetTrackAnimationSet( 0, NULL );

            // We're done
            return;

        } // End if not keyframed

        // We're done with the normal animation set
        pAnimSet->Release();
```

After fetching the animation set, we make sure it is a keyframed animation set, since those are the only ones we know how to subdivide in this tool. If the selected animation set is not a keyframed set then we once again disable the marker and slider controls and clear track 0 on the animation mixer.

If we get this far, then the user has selected a valid source animation set. We set the animation set to track 0 on the mixer (ready for playback) and cache the animation set interface and its index in the member variables of the dialog object. We then enable the marker and slider controls so that the user can begin to use the slider bar and set their markers. Until an animation set is selected, these controls are ghosted and cannot be used.

```
        // We're using the keyframed animation set
        m_pActor->SetTrackAnimationSet( 0, pKeyAnimSet );

        // Store the key anim set, and don't release this one!
        m_pCurrentAnimSet = pKeyAnimSet;

        // Store the index too for later
        m_nCurrentAnimSet = nAnimSet;

        // Enable the animation set controls
        EnableSetControls( true );
```

There is one final job to do before we leave this function. When configuring the properties of a split definition, we use the slider to set two markers. We also have to assign that split definition a name using the Set Name edit box. When the user changes the source set, we will copy into this edit box the name of the source set by default. The user can change it, and will most likely want to, but if they are simply trimming an animation set, then this saves them from having to re-type the name of the new animation set when they wish to keep it the same as the original.

So before we exit this function, we fetch the name from the animation set currently selected and copy it into a temporary buffer. If the animation set has no name then we will simply fill this buffer with the string “Unnamed Animation Set”. We then get the handle of the Set Name edit control and set its text to the contents of this buffer.

```
        // Set the text for the new set name
        TCHAR   NameBuffer[512];
        LPCTSTR strName = pKeyAnimSet->GetName( );
        if ( strName == NULL || _tcslen( strName ) == 0 )
            _tcscpy( NameBuffer, _T("Unnamed Animation Set") );
```

```
        else
            _tcscpy( NameBuffer, strName );

        // Get the definition information box
        if ( !(hControl = GetDlgItem( m_hDialog, IDC_EDIT_NEWSETNAME )) ) return;
        Edit_SetText( hControl, NameBuffer );

    } // End if selected a legitimate item
}
```

CAnimDialog::AddSetDefinition

This method is called after the user has positioned the start and end markers, set the new name, and wants to add this new definition data to the list of set definitions. The first section of code just makes sure that the user does not try to add a large number of definitions. It prevents overflow of the dialog's internal array of animations which is allocated at application startup to be MAX_DEFINITIONS in size. This is set to 256 by default. This means that this tool allows you to break a single set into 256 sets. Feel free to change this value in the source code if you need more.

```
void CAnimDialog::AddSetDefinition()
{
    HWND  hControl = NULL;
    TCHAR NewName[128];
    ULONG i;

    // Validate there are enough slots left
    if ( m_nDefCount >= MAX_DEFINITIONS )
    {
        ::MessageBox( m_hDialog, _T("You have reached the maximum number of
                                    animation set definitions supported by
                                    this application."),
                                _T("Limit Reached"),
                                MB_OK | MB_ICONSTOP );
        return;

    } // End if limit reached
```

This next section of code tests that a source animation set is currently selected since each split definition must contain a source animation set. If a source animation set is not selected when the user presses the Add Definition button, then this split definition should fail to be added to the list. This should never actually happen because those controls should be disabled if no animation set is selected, but let us be cautious about this.

```
    // Validate that an animation set is available
    if ( !m_pCurrentAnimSet )
    {
        ::MessageBox( m_hDialog, _T("You must specify a source animation set to
                                    extract data from."), _T("Invalid Source"),
                                    MB_OK | MB_ICONEXCLAMATION );
        return;
    } // End if no source
```

We will also fail if the start and end markers are equal. This would essentially cause the creation of an empty animation set, which would serve no prupose.

```
    // Validate that some range of data is selected
    if ( (m_nEndTick - m_nStartTick) == 0 )
    {
        ::MessageBox( m_hDialog, _T("The data range specified is empty,
                                    you must select a portion of the animation
                                    to extract."), _T("Invalid Range"),
                                    MB_OK | MB_ICONEXCLAMATION );
        return;

    } // End if no source
```

Next we make sure that the user has supplied a name for the definition. We want to enforce this behavior so that all of our new animation sets will have unique names. This is especially important since they are displayed in the list boxes using their names. So we fetch the handle of the Set Name edit control and extract its text. If the length of the retrieved text is zero, the edit box is empty and we fail to add the definition to the list.

```
    // Get the new set name
    if ( !(hControl = GetDlgItem( m_hDialog, IDC_EDIT_NEWSETNAME )) ) return;
    Edit_GetText( hControl, NewName, 128 );

    // Empty ?
    if ( _tcslen( NewName ) == 0 )
    {
        ::MessageBox( m_hDialog, _T("All new set definitions must have a name."),
                     _T("Invalid Option"), MB_OK | MB_ICONEXCLAMATION );
        return;

    } // End if no name
```

When working with text mode X files, there are certain characters reserved by the specification for the purposes of structuring the data. These characters include semi-colons and curly braces. In certain circumstances, when we include such reserved characters in animation set names, we can actually inadvertantly corrupt the integrity of the file.

For this reason, we introduce the following loop, which forcably removes any non-alphanumeric characters from the animation set name and replaces them with an underscore. This prevents the problem from occuring.

```
    // Replace all non numeric / alpha characters with underscores
    // (the text .X file definition doesn't seem to like much else)
    for ( i = 0; i < _tcslen( NewName ); ++i )
    {
        TCHAR ch = NewName[i];
        if ( (ch < _T('A') || ch > _T('Z') ) &&
             (ch < _T('a') || ch > _T('z') ) &&
             (ch < _T('0') || ch > _T('9') ) )
        {
            // Replace
            NewName[i] = _T('_');
```

```
        } // End if invalid character
    } // Next Character
```

Now we are almost ready to add the definition information to the array. However, we must make sure that a split definition does not already exist with the same name. We enforce (for the same reasons mentioned above) the rule that all definitions must have unique names. In the next section of code, we loop through each split definition in the array and compare its name with the name of the new definition we are about to add. If the _tcscmp function returns zero, the strings are the same and we have found that a split definition already exists with the same name as the one we are about to add to the list. When this is the case, we fail to add the split definition.

```
    // Search to see if a name matching this already exists
    for ( i = 0; i < m_nDefCount; ++i )
    {
        if ( _tcsicmp( m_pDefList[i].SetName, NewName ) == 0 )
        {
            ::MessageBox( m_hDialog, _T("A new set definition with this name
                                        already exists in the list. Names must
                                        be unique."),
                                        _T("Invalid Option"),
                                        MB_OK | MB_ICONEXCLAMATION );
            return;

        } // End if matching name found

    } // Next Definition
```

If we get this far, the marker positions, the source animation set, and the name we have assigned to this new set are all valid, so we will copy the information into the array. We already have the new name stored in a local variable and the currently selected source animation set index was also cached in the member variable when the the user selected the animation set (see previous function). Also, when the user selects the Set Range Start and Set Range End buttons, the handler functions cache the current position of the slider in the m_nStartTick and m_nEndTick member variables. Therefore, the only thing left to do is add the information to the end of the definitions arrays and increase the definition count. We then call the PopulateDefinitions function to refresh the split definitions list box so that the new addition is now shown.

```
    // Populate the new definition item
    _tcscpy( m_pDefList[ m_nDefCount ].SetName, NewName );
    m_pDefList[ m_nDefCount ].SourceSet  = m_nCurrentAnimSet;
    m_pDefList[ m_nDefCount ].StartTicks = m_nStartTick;
    m_pDefList[ m_nDefCount ].EndTicks   = m_nEndTick;

    // Increment
    m_nDefCount++;

    // Repopulate Definition Items
    PopulateDefinitions();
}
```

CAnimDialog::ApplyChanges

The final method that we will examine in this lab project is the one that actually creates the new animation sets from the split definitions list. Although this might seem like it would be a large function, it actually just calls the CActor::ApplySplitDefinitions method to do all the work.

The function returns if no valid actor is available or if no definitions have yet been added by the user.

```
void CAnimDialog::ApplyChanges()
{
    // Bail if there is no actor
    if ( !m_pActor ) return;

    // Ensure there are any definitions to apply
    if ( m_nDefCount == 0 )
    {
        ::MessageBox( m_hDialog, _T("There are no definition changes to be
                                     applied."),
                                    _T("Empty Definition"),
                                    MB_OK | MB_ICONEXCLAMATION );
        return;

    } // End if no name
```

Now we know we have a valid actor and at least some definitions in our array, so we will release the currently selected animation set interface. This is because this animation set is about to be destroyed and replaced by the new ones described by the split definitions. We then call our CActor member function to create the new sets and release the old ones.

```
    // Release the currently selected animation set, it's about to be destroyed.
    if ( m_pCurrentAnimSet ) m_pCurrentAnimSet->Release();
    m_pCurrentAnimSet = NULL;
    m_nCurrentAnimSet = 0;

    // Apply the split definition
    m_pActor->ApplySplitDefinitions( m_nDefCount, m_pDefList );

    // Total re-population please
    UpdateDialog();
}
```

Finally, at the end of the function we call the CAnimDialog::UpdateDialog method to refresh all the controls with their new information. This will empty the definitions array and repopulate the top list box with the new list of source animation sets (the ones we have just created). We now have a new list of source sets and an empty array of definitions (as they have all be used and discarded). Usually, at this point you will want to save the actor back out to an X file in its new format.

Conclusion

This workbook has been one of our largest to date. In the process, we have learned alot of new material and have simultaneously turned our CActor class into a powerful tool. It now encapsulates X file loading and saving for complete multi-mesh hierarchies that include animation data. We have also exposed the powerful features of the D3DX animation controller through our CActor interface. Finally, in Lab Project 10.2 we developed a utility that better educates us about how the actor might be used in a non-game application. It is also a tool that will be extremely useful for breaking an animation set with combined sequences into multiple animations sets for the purposes of our game engine.

In the next two chapters we will extend the actor even further when we learn how to perform character skinning using both software and hardware techniques. Make sure that you are comfortable with everything we have covered thus far, as it will all be used again in the next few lessons.

Chapter Eleven

Skinned Meshes and Skeletal Animation I

Introduction

Newly emerging graphics technologies have allowed gamers to enjoy scene realism that would be unthinkable only a few years ago. With new cards now supporting some 256MB or more of on-board local video memory and features like real-time decompression of textures during rendering, we now have the ability to use greater quantities of higher resolution textures. We have also seen a dramatic increase in polygon budgets due to similar advances in vertex processing hardware. The combined effect is that the game artist experiences a larger degree of creative freedom coupled with the confidence that the game engine can handle the loads in real time. As a result, the virtual worlds we take for granted in modern computer games can often look so real as to be almost photo-realistic. But in the area of animation, and specifically with respect to game characters, until fairly recently the technology has not kept pace. Traditionally, as soon as you saw a character animating in the scene it was obvious that this was not a photo or a movie you were looking at.

In early games, comparatively low CPU speeds and the lack of dedicated 3D transformation and lighting hardware meant that per-vertex processing had to be kept to a minimum in order to maintain an acceptable frame rate. In-game characters were often constructed as segmented polygonal objects, where each limb of the character would be created as a separate mesh and attached to a frame hierarchy. The frames of the hierarchy represent the 'bones' of the character. Animations like walking or shooting could be applied to the hierarchy (as we examined in Chapter Ten), causing the meshes of the separate body parts attached to the individual frames to move. While this approach achieved its objective, it brought with it some undesirable artifacts. Because each body part was a separate mesh, when two neighboring body parts (such as an upper leg mesh and a lower leg mesh) were rotated at extreme angles with respect to each other, gaps between them became very obvious. Unless extreme care was taken on the modeling side to reduce this artifact – and it could not always be successfully done – the results were less than ideal.

As CPU speeds increased, a greater degree of per-vertex processing could be done in software and a technique called *vertex blending* emerged. Vertex blending is the process of transforming a model space vertex into world space using a weighted blending of multiple world space matrices. This is in contrast to the single world matrix concept we are familiar with. This technique allows programmers to use a single mesh for an entire character while retaining all of the benefits of a hierarchy for animation (much like the segmented character example mentioned previously). The single mesh behaves like a "skin" draped over the "bones" of the hierarchy such that when the bones (i.e. the frame hierarchy) are animated, rather than just rotating limbs constructed from separate meshes, the entire skin (i.e. mesh) is deformed -- bending and stretching to maintain the shape described by the underlying skeleton. Because the skin is a single mesh, no cracks will appear and the limbs of the character smoothly bend and stretch to provide much more realistic animation.

To make this skin/bone concept clearer, first consider a segmented object, where each mesh is assigned to a single frame in a multi-frame hierarchy. Forget about the fact that each mesh has its own vertices and forget about which meshes these vertices actually belong to. Based simply on the frame the parent mesh was originally assigned to, you can imagine that some vertices would be animated by one frame matrix, while other vertices belonging to another mesh assigned to another frame would be animated by a different frame matrix. If we were to go one step further and imagine that all of the vertices in the

hierarchy were actually a part of a single mesh and we add the ability to specify, on a *per-vertex* level, which matrix in the hierarchy should be used to transform each vertex, then we wind up with our skin concept. That is, we have a single mesh (a skin) animated by the frame hierarchy (a skeleton) such that each vertex in the mesh stores information about which matrix in the hierarchy transforms it to world space. If we were to play a walk animation on our hierarchy, then we would see that when the vertices that are assigned to the leg frames (bones) are animated, the legs of the character mesh can bend without causing cracks or visible seams.

Although assigning a single vertex in the mesh to a single frame in the hierarchy would cause the skin to deform, often this is too restrictive for producing realistic skin animation. This is where vertex blending comes in. To see where vertex blending can play a role, look at the skin around your elbow and imagine it as a mesh of vertices. If you bend your arm at the elbow, retracting the forearm up to meet your shoulder (imagine that your forearm is a bone in the hierarchy), you can clearly see that the skin at your elbow is affected by this bone movement. The skin changes to cover the new shape formed at the elbow joint. Thus we might say that the vertices of your elbow should be transformed by the forearm frame matrix in the hierarchy. That is, whenever the forearm frame is rotated, the elbow vertices that are children of that frame would be directly manipulated by that frame's matrix.

However, if you stretch out your arm in front of you and just rotate your wrist up and down, you can see that while not as severe as the previous example, the rotation of the wrist bone also has some influence on the skin near the elbow -- causing it to shift and change slightly. We could say then that the elbow vertices are directly affected by the rotation of the forearm bone and also by the rotation of the wrist bone (and theoretically other bones as well, such as the shoulder bone). Therefore, to correctly model the behavior of our skin, we would need to be able to specify not only one matrix per vertex, but in fact any number of matrices per vertex. While some vertices in the mesh may only be affected by a single bone, others may be affected by two or more different bones to varying degrees, sometimes simultaneously. Although the orientation of your wrist bone did affect the position of the skin around your elbow, it did not influence the skin position to the same degree that rotating your forearm did. Therefore, to effectively skin a mesh:

1. We need a **skeleton**. This is just a frame hierarchy where each frame is assumed to be a bone in the skeleton. Each frame matrix is referred to as a bone matrix.
2. We need a **skin**. This is just a standard mesh describing the character model in its reference (default) pose. The vertices are defined in model space just like any other standard mesh.
3. We need a **vertex/bone relationship mapping**. We want to know for each vertex, which bones (frames) in the skeleton (hierarchy) directly affect it. The bone(s) will ultimately be used to calculate the world space position of the vertex. As described, a single vertex may be directly connected to several bones, such that movement of any of those bones will influence its final world space position to some degree.
4. We need **weights** for each vertex describing how influential a particular bone will be in calculating its final world space position. If a vertex is influenced by N bones, then it should also contain N floating point weight values between 0.0 and 1.0. These describe the influence of that bone on the vertex as a percentage. For example, if a vertex stores a floating point weight value of 0.25 for matrix N, it means that the transformation described by bone N (the frame matrix) should be scaled to 25% of its strength and then applied to the vertex. So if a vertex is influenced by N matrices, then it should store N weights. The sum of the N weight values should add up to exactly 1.0.

So now we need the ability to assign individual vertices, rather than entire meshes, to frames in our hierarchy. We also need the ability to perform vertex blending. Item four in the previous list is where vertex blending comes in. We know that the first thing the DirectX transformation pipeline will typically do is transform vertices into world space using the currently set world matrix (a single matrix). Things are a little different now however because, for any skin vertex currently being transformed, a vertex might need to be transformed into world space using contributions from a number of bone matrices. This is how vertex blending works. If we were going to implement vertex blending ourselves (not using APIs like DirectX or OpenGL), then our code would have to include a vertex blending function. That function would be called for each model space source vertex and generate a world space destination vertex that had been transformed using the contributions of several matrices. Vertex blending is actually a remarkably easy thing to do as we will briefly describe below.

> **Note**: The DirectX pipeline will perform vertex blending on our behalf, but an understanding of the blending process is essential to understanding skinning. This next section is intended to show you how vertex blending works and what is going on behind the scenes.

Imagine that we wish to perform vertex blending with three matrices for a given model space vertex. In this example we can think of the model space vertex as being a vertex in the mesh which is assigned to three bones in the skeleton. Remember that bones are ultimately just frame matrices and the skeleton is just a frame hierarchy.

First we take the model space vertex and, in three separate passes, transform it by each world space bone matrix to create three temporary vertices:

ModelVertex = Model Space vertex we wish to transform into world space

M1 = 1st World Space matrix to use in blending
M2 = 2nd World Space matrix to use in blending
M3 = 3rd World Space matrix to use in blending

```
TempVertex1  = ModelVertex * M1;
TempVertex2  = ModelVertex * M2;
TempVertex3  = ModelVertex * M3;
```

At this point we are midway through the process. When performing vertex blending, each vertex also stores some number of weights (sometimes referred to as *beta weights* or *skin weights*) with values between 0.0 and 1.0. These weights describe how influential a specific matrix will be when determining the final position of the world space vertex. For example, if a vertex is to be blended into world space using three matrices as above, the vertex would contain three weights describing how strongly the transformation in each matrix will contribute to the final vertex position. If we were to sum these weights, they should add up to 1.0. If the vertex used in the above example had weight values of 0.25, 0.5, and 0.25, then we know that the matrix M1 would contribute 25%, matrix M2 would contribute 50%, and matrix M3 would contribute 25% to the final vertex. Therefore, after we have generated the temporary vertex for each matrix, we can scale each temporary vertex position by the corresponding weights. The second part of our vertex blending procedure would do just that:

TempVertex1 = TempVertex1 * ModelVertex.Weight1; //*(Weight1 = 0.25)*
TempVertex2 = TempVertex2 * ModelVertex.Weight2; //*(Weight2 = 0.5)*
TempVertex3 = TempVertex3 * ModelVertex.Weight3; //*(Weight3 = 0.25)*

Now that we have scaled each temporarily transformed vertex position by the corresponding weight, we sum the results to create the final blended world space vertex position.

Blended Result = TempVertex1 + TempVertex2 + TempVertex2;

That is all there is to vertex blending. Note that although it is not *required* that the weights add up to 1.0, there may be strange and unpredictable results when they do not. We generally want our fractional transformations to add up to one complete transformation.

The following code gives us a bit more insight into the vertex blending process. It takes a model space vertex, an array of matrices, and an array of vertex weights (one for each matrix in the array) and returns a final blended world space vertex.

```
D3DXVECTOR3 TransformBlended (D3DXVECTOR3 *pModelVertex ,
                              D3DXMATRIX *pBlendMatrices ,
                              float * pWeights , float NumOfMatrices )
{
      D3DXVECTOR3 TmpVertex;
      D3DXVECTOR3 FinalVertex ( 0.0f , 0.0f , 0.0f );

      for ( int i = 0 ; i < NumOfMatrices; i++ )
      {
            TmpVertex = ( *pModelVertex * pBlendMatrices[I] ) * pWeights[I];
            FinalVertex += TmpVertex;
      }

      return FinalVertex;
}
```

Vertex blending is the key process behind character skinning and skeletal animation. With vertex blending, we can state that some vertices in the mesh are connected to more than one bone in the hierarchy, like our elbow vertices as discussed previously. In such a case the vertices in this area might be attached to both the upper arm bone and the lower arm bone. Since each bone is just a frame in the hierarchy, we can transform the vertices in this region of the mesh into world space simply by performing a blend (as shown above) using the matrices for both the forearm bone and the upper arm bone. When either or both of these bones are animated, the vertices of the skin move and change shape to accommodate the new bone positions, much like our skin does in real life. Because all of the vertices in the skin are a single mesh, we do not suffer the crack problems between the joints of the character's limbs. We will see later that it is the job of the 3D artist to construct the bone hierarchy and assign vertices in the mesh to these different bones. The artist will also be able to alter the weights of each vertex to tweak the model and get the weights of all influential matrices for a given vertex just right.

From the programming perspective, one of our first jobs will be to load the skeleton. Fortunately, we already know how to do this because a skeleton is just a D3DXFRAME hierarchy loaded automatically

by the D3DXLoadMeshHierarchyFromX function. Each frame in the hierarchy will be a bone and its frame matrix is called a *bone matrix*.

> **NOTE:** While we are now referring to the frame hierarchy as a *skeleton* and the frames as the *bones* of this skeleton, nothing has actually changed with respect to chapters 9 and 10. This is just a naming convention we are using because it is more appropriate under these circumstances.

Most gamers were quite impressed the first time they saw the skeletally animated skinned mesh characters in the original Half-Life™. Their smoothly deforming skins set a precedent that could not be ignored. Pretty soon, almost all new games were implementing character skinning and skeletal animation as the technique of choice, and the characters in computer games have become a lot more believable as a result. Keep in mind that skinning techniques are not reserved exclusively for character animation only. One of the lab projects in this chapter will use skinning and skeletal animation to create some very nice trees for our landscape scenes. Such a system works quite nicely for this purpose and you will soon be able to play wind animations that cause swaying branches and rustling leaves.

While increased CPU speeds meant that vertex blending could be performed in real time, things are even better today since vertex blending can now be performed by the GPU. This is true for most T&L graphics cards released since the days of the first geForce™ cards. The DirectX 9.0 API exposes these features to our application so that we can perform hardware accelerated vertex blending on the skin and bones system that we implement.

There are a number of additional advantages to using a skin and bones system beyond the removal of cracks and seams discussed earlier. First, the artist has to do very little repetitive mesh building. The artist only has to create a single mesh in its ‘reference pose’ and then use the modeling package of his/her choice to fit a skeletal bone structure neatly inside the mesh. After assigning which vertices will be influenced by which bones and specifying the appropriate weights, the main work is done. From that point forward it is a matter of applying any animation desired to the skeleton to see the results.

Another great thing is that many of the more popular commercial modeling packages export the skin and bones data into X file format. The mesh is stored as a regular mesh and the skeleton is just stored as a vanilla frame hierarchy (like the ones we looked at in previous lessons). Later in the chapter we will examine all of the X file data in detail.

From our perspective as programmers, we benefit from the fact that animating the bones of a character is no different from animating a normal frame hierarchy. Thus our last chapter will serve us well as we work through our lab projects. We will still use the animation controller to choose the animation we wish to play and call AdvanceTime to update the frame matrices in the hierarchy. Once the matrices have been updated, we traverse the hierarchy and build the absolute world matrices for each frame (bone). Then all that is left to do is set the matrices and render our mesh.

Transforming and rendering the mesh is where things will change a fair bit. We must now be able to set multiple world matrices on the device so that the transformation pipeline can calculate the world space position of each vertex using all of the world space bone matrices to which it is connected. We will also see how the vertex structure of the mesh has to change so that it can store a number of weights (one for each bone matrix that affects it). Once the device has this information however, we can call DrawPrimitive as usual and let the pipeline render each subset. The device will use the multiple matrices

we have set from our bone hierarchy to perform the vertex blending. This is certainly going to be exciting material for sure, but we do have a fair bit to learn about before we can perform this process properly.

We have now had a rapid tour of what skinning and skeletal animation is and why it is the technique of choice for most game programmers. Very little code is needed to implement it, and it takes up very little memory compared to other techniques (such as tweening, as we will see next). It also gives us control over every aspect of the character's body. You can instruct the artist to create bones that can be moved to cause wrinkles in the skin, twitch the character's nose, or cause the characters hair to blow in the wind. These are all very cool animation effects that we are seeing in today's computer games and now we are going to learn the foundations for how to do achieve this. But first, we will take a very brief diversion…

11.1 Vertex Tweening

There are alternatives to skinning that can produce very good results. One popular choice is called **tweening** (or **morphing**). id Software® used this technique in the Quake II™ engine to produce smooth character animations without cracks or other visual artifacts.

Tweening is similar to the keyframe animation approach we discussed in our last lesson. In the case of Quake II™ for example, the file (.md2 file in this instance) stores a number of keyframes, but these keyframes do not store a timestamp and transform (SRT) data. Instead they store a timestamp and an entire mesh in a given pose. Each mesh represents exactly what the vertex data of the mesh should look like at the specified time stamp. The file will therefore contain a large list of pre-posed meshes.

Let us imagine an .md2 file containing a 'walk' animation cycle. The file will store a number of meshes, where each mesh can be thought of as a snapshot of the character at a certain time throughout the life of the walk animation.

To render the mesh for a given time (say T = N + 0.5 -- where N is an arbitrary time through the duration of the animation) the first thing we would do is find the two keyframes in the mesh list that bound this time. In this example it would be keyframe N and keyframe N+1. Our two keyframes would now be two separate meshes of the character in different poses as shown in Fig 11.1.

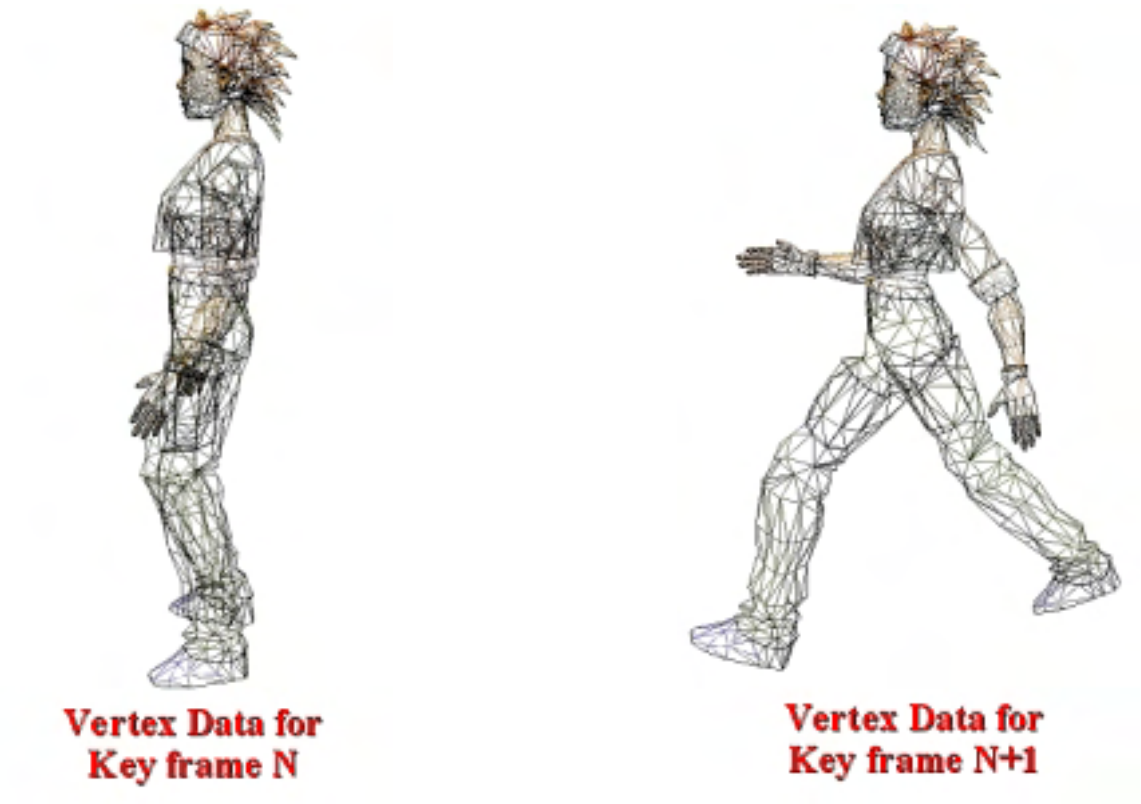


Figure 11.1

To calculate the final position of each vertex in the mesh we will interpolate between matched vertices in both sets. We will use the amount that the requested time is offset between the time stamps of both keyframes as the interpolation weight factor. So we are actually building a new mesh on the fly by interpolating between the vertex sets of two input meshes. The interpolation would look something like the following pseudo code:

```
t = AnimTime – KeyFrame[N].Time;

for( I = 0 to NumVertices )
{
     Vertex V1 = KeyFrame[N].Vertex[I];
     Vertex V2 = KeyFrame[N+1].Vertex[I];

     FinalMesh.Vertex[I] = V1 * (1.0 – t)  +  V2 * t
}
```

In the current example, AnimTime is the time we request to render the mesh and N and N+1 are assumed to be the indices of keyframes that bound the requested AnimTime. Since each mesh is simply the same object in different poses, we know that there will be the same number of vertices in both of the source meshes. We also know that there will be exactly this many vertices in the dynamically generated output mesh.

If we imagine that we would like the mesh generated for an animation time of N+0.5 (midway between the two keyframe poses above) we would have a temporary vertex buffer containing the interpolated vertex data. The mesh would look like the results in Fig 112.

Figure 11.2

We could now transform and render this mesh just as we would any other standard mesh.

So as you can see, during each frame we generate a new model space mesh by interpolating between keyframe meshes. Once we have the vertices for both of the source meshes, we interpolate between them to generate the in-be***tween*** pose, which is where the term ***tween***ing originates.

While this is a simple technique to implement in software and it does allow us to use single skin meshes, it does have its drawbacks. The main problem is that it can consume a good deal of memory because we have to store multiple copies of the mesh vertex buffer (one for each key pose).

DirectX 9 exposes hardware support for tweening and many graphics cards support the acceleration of tweening on the GPU. However, we will not cover tweening in this chapter for a number of important reasons. First, we cannot implement tweening without using declarator-style vertex formats (non-FVF) and these will not be discussed until Module III in this series . Second, many programmers have now abandoned tweening/morphing as a means of generically animating characters in their games in favor of skin and bone systems (mostly due to the memory costs). However, that is not to say that vertex tweening is without value -- far from it. Tweening is still a very popular technique in the area of facial animation. With the release of Half-Life 2™, facial animation is certainly something that is on the minds of game developers everywhere. An artist can build multiple facial expressions that can be easily morphed from one to the other to add lots of additional personality to a game character. However, it is worth noting that skin/bones can also be used for this purpose and is rapidly gaining popularity. Tweening will be covered in a little later in this course series, so do not worry too much about it for now.

11.2 Segmented Character Model Animation

For completeness, before we work our way up to skinned meshes and skeletal animation we will first look at how we might perform character animation the old fashioned way -- by using what we learned in the previous chapter on hierarchical animation. This is the way characters used to be implemented in the games of a few years ago. This was because per-vertex processes (such as vertex blending) were too expensive to be performed in real-time due to the low CPU speeds and the lack of 3D hardware acceleration in most end user systems. Taking a look at this older approach however should help us understand the origins of skinning as well as why skeletal animation and skinning are such useful techniques to have in our tool chest.

Armed with only the animation knowledge from the previous chapter and instructed to create an animated character that can play a number of animations (such as walk, run, and jump for example), we might imagine that we could use the exact same system from Lab Project 10.1. We would perhaps instruct our game artist to build a character object using multiple mesh body parts and arrange them in a hierarchy inside the modeling program. This is no different from how we constructed the space scene in the previous lesson. We had multiple meshes (the space ship, the bay doors, and the radar dishes) each attached to frames in a hierarchy and it all worked nicely. The fact that we are now using the hierarchy to represent meshes of human body parts should theoretically work just as well.

For the sake of simplicity, let us work only with the bottom half of a character. This lower body might consist of five separate meshes: a pelvis, a left upper leg mesh, a left lower leg mesh, a right upper leg mesh and a right lower leg mesh. The five separate meshes might be arranged in the hierarchy as shown in Figure 11.3.

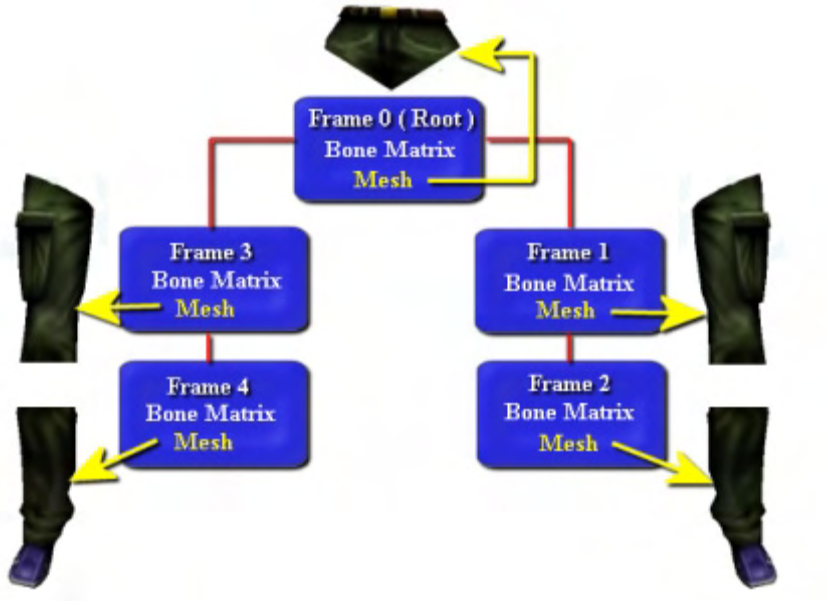


Figure 11.3

You should already have a good idea about how this frame hierarchy would be exported and stored in an X file. Certainly there would be little trouble loading it in using D3DXLoadMeshHierarchyFromX as we learned in previous lessons. Because we are now using the frame hierarchy to represent a character instead of a generic scene, we will refer to each frame in the hierarchy to which a body part is attached as a 'bone'. This works well because they will behave very much like a set of human bones. That is, when we rotate one of our bones in real life, our skin and clothes around that bone move also. Likewise, when we rotate a frame in our hierarchy, a single body part attached to that frame will move (along with any child frames and their respective meshes). So we will think of each frame as an individual bone and the frame hierarchy as a skeleton. In Fig 11.3, we can see that the pelvis is the root bone and that it has a pelvis mesh child. It also has two child bones which are the upper parts of each leg (along with their meshes). Each upper leg bone also has a lower leg child bone (each of which has a lower leg mesh).

After the artist has constructed the hierarchy of meshes in Fig 11.3 he could create a series of appropriate animation keyframes for walking, running, crouching, etc. and export everything to an X file. From our perspective, nothing has changed. We can still use D3DXLoadMeshHierarchyFromX to load this character model hierarchy using exactly the same code we used in the last chapter to load our scene (a scene of body parts now). We will get back a fully loaded hierarchy along with an ID3DXAnimationController so that we can start assigning the different animation sets (walk, run, etc.) to tracks on our animation mixer. To play a given animation requires only a call to ID3DXAnimationController::AdvanceTime, just as we saw in our last two lab projects.

So it seems as if we have achieved character animation free of charge simply by re-using the code we already have. However, as discussed in the introduction, this approach is not without its problems. The obvious problem with representing characters using the segmented character model is that in real life our various limbs are not separate detached entities. While it is true that our bones most definitely are individual segments of our skeleton and that they can move independently, our skeleton is covered with muscle and a continuous layer of skin which behaves very differently. Because this is not the case with our segmented character representation, certain artifacts which do not arise in real life become unavoidable.

Fig 11.4 shows one of the legs in our segmented character. We know already that the leg consists of two bones and two meshes -- the upper leg mesh is attached to the Frame 1 bone and the lower leg mesh is attached to the Frame 2 bone. These are the frame matrices in their default pose (commonly referred to as the *reference pose*) where the character meshes have been carefully arranged so that no cracks or spaces can be seen between the two separate leg meshes. The red dashed line in Fig 11.4 shows the boundary where the two leg meshes meet.

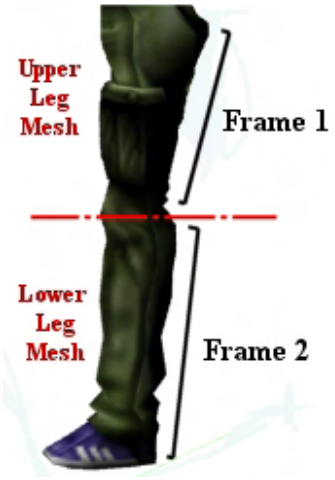


Figure 11.4

If we were to rotate Frame 1 (which we can think of as the joint at the top of the upper leg mesh where it meets the pelvis) relative to its pelvis parent, the leg would rotate without any artifacts. The lower leg is a child of the upper leg, so it too would be rotated. The entire leg would rotate much like the hand of a clock. However, if we were to rotate just the Frame 2 bone, (located in the knee area at the center of the leg) we would want the leg to bend in the middle (required if we want to play a realistic walking or running animation). But Fig 11.5 shows us what happens to the segmented character model system if the Frame 2 bone is rotated (thereby rotating its attached mesh).

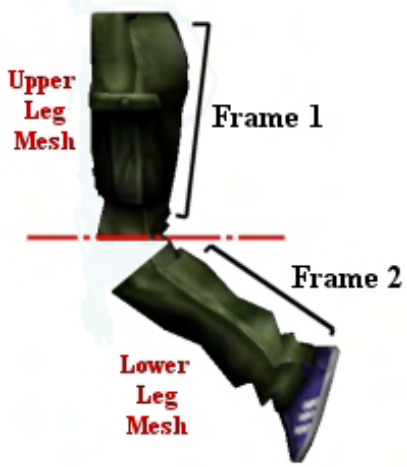


Figure 11.5

Fig 11.5 makes it instantly clear that the leg mesh is not one smoothly skinned entity, but is in fact two meshes. While this artifact can be minimized to some degree by careful modeling of the separate body part meshes (ex. tapering and overlapping them where they meet so that the crack is not so obvious) this is not something we can entirely avoid using the segmented character model. The problem is that each mesh is also being treated exactly like its associated bone. If we were to remove the skin from our own bodies, we would see that our skeletons would suffer the same artifacts (where each bone would be connected by a ball joint for example but would appear quite distinct).

So what are we to do to remedy this problem? Tweening would be one possible solution, but we have already discussed why it might not be the best choice for us. Beyond the memory concerns, consider as well that we spent a good deal of time learning how to animate hierarchies and use the D3DX animation system. We have already experienced the power and flexibility of the animation controller and it would be a shame to have it stripped away from us. Ideally we want a system where we can still store and play our animations using the same techniques we are already comfortable with but that addresses the concerns discussed thus far. This means that we want to represent our characters using a bone hierarchy (just like the previous example) but we do not want to use a segmented geometry model.

> **Note:** One final point in favor of using a hierarchical bone structure over a tweening system that is worth thinking about is that tweening is quite restrictive. Essentially we are locked into having to interpolate frames provided by the artist. With a skeletal structure, while we will most often be playing back animation created by the artist in a modeling package, we still have the freedom to programmatically apply transformations to any bones in the hierarchy we see fit (even at run time). In that sense, there are an infinite number of ways that we can animate a skeletal structure.

11.3 Skinning

Mesh skinning, used in conjunction with a skeletal structure, provides the solution to the problems previously discussed (although it is not without its own issues). The skeleton remains the same – it is a frame hierarchy and we can generate and play back SRT keyframe animations exactly as we did before. But instead of assigning separate meshes to individual frames, we are now going to use a single mesh, referred to as a *skin*. That is, rather than assign different meshes to different bones we will instead assign different *vertices* within a single mesh to different bones.

In the simplest case, each vertex in the skin is assigned to only one bone. We could theoretically imagine the skin as being constructed from separate sub-meshes, where each sub-mesh is a group of vertices attached to the same bone. The difference is that all sub-meshes share a single vertex buffer because they are all part of the same mesh object. In the more complex (and probable) case however, the logical division of the mesh into imaginary subsets is not so clear because we can assign a single vertex to more than one bone in the hierarchy. When a vertex is to be transformed, several bone matrices are used (to varying degrees) to determine the final world space position of that vertex. When a bone is rotated, any vertices that are assigned to that bone will be rotated also. The degree to which they will rotate will depend on the weights specified for that bone on each vertex. For the moment, do not worry about how the mesh is stored in the hierarchy or how the vertices are attached to frames, we will get to that a little later in the chapter.

While the artist is responsible for generating this mesh and constructing the bone hierarchy such that it correctly describes the mesh in its reference pose, the artist will also be responsible for determining which bones in the skeleton will influence each vertex, and by how much (using the weight value). This is not as hard as it might seem and we will discuss creation of skin and bone data from the artist's perspective in just a moment.

Fig 11.6 depicts the leg section of our single skin as it might look in our skinning application. The bones still exist in the same places since the skeleton is unchanged from our previous segmented example (Fig

11.6). The leg mesh depicted here however is now part of a single unified skin. In this example we are focusing on the knee joint (the Frame 2 bone). Most of the vertices in the lower leg section of the skin might be assigned to just the lower leg bone (Frame 2) and most of the vertices in the upper section of the leg might be mapped to just the upper leg bone (Frame 1). However, the vertices about the knee joint (within some limited radius) are probably going to need to be influenced by both bones.

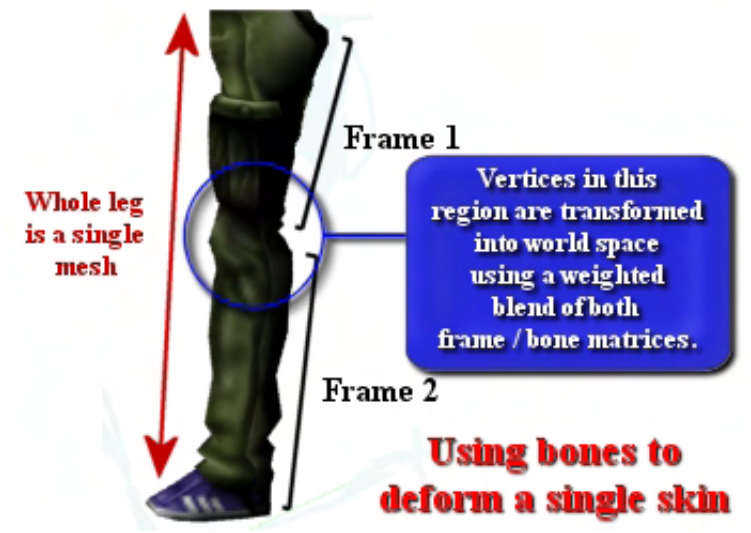


Figure 11.6

Because this is a single mesh, when the knee bone is rotated (Frame 2) all of the vertices attached to that bone will also move by some amount determined by the weight stored in the vertex for that bone. The vertices outside the blue circle in the lower part of the leg will be 100% influenced by this bone because they are attached to only the Frame 2 bone. However, the vertices in and around the knee joint are also influenced to some degree by both bones, so their positions will change also but not quite to the same extent. As the vertices in this area move, they stretch the skin polygons of which they are a part so that they occupy the space that would have caused a gap in the segmented character model. We see in Fig 11.7 that this rotation causes no cracks to appear -- the section of the skin around the knee joint smoothly deforms to give the impression of a single skin covering our skeleton.

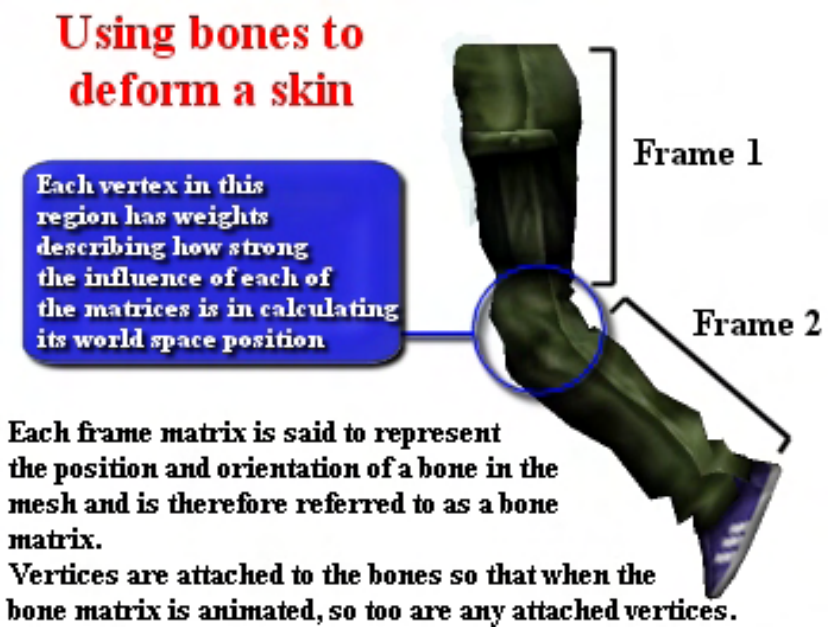


Figure 11.7

The result in Fig 11.7 is much more realistic than our earlier segmented character results.

As mentioned earlier, DirectX 9.0 supports hardware acceleration of the vertex blending process used to transform each vertex from model space into world space. There is support for using up to four different bone matrix influences per vertex, which is generally more than enough. Since many of the more recent graphics cards provide acceleration of vertex blending in hardware, skinning becomes an exciting prospect. Further, even if hardware support for skinning is not available on the user's machine, we can instruct DirectX to use software vertex processing to transform our skins. In this case the vertex blending will be carried out in the software module of the pipeline. Unlike some of the software processing alternatives supported by the DirectX pipeline, the software vertex blending module is very efficient and can be considered extremely useful even in a commercial project. You may even find that on some systems (depending on the video cards and CPU speeds) there is very little performance difference between hardware accelerated blending and software vertex blending performed by the Direct X transformation pipeline (although to be clear, this is not typically the case). The bottom line is that with or without hardware acceleration, we can still perform real-time skinning.

11.3.1 Generating Skinned Mesh Data

We already know how to animate a skeletal hierarchy but we still have no idea how the skin will be represented in memory. Nor do we know how to let the transformation pipeline know which vertices in the skin should be influenced by which matrices in the hierarchy and what weight values should be used. Rather than dive straight into API specific function calls, let us start with an analysis of the data first. It will be helpful to start at the beginning of the process and briefly discuss the steps the artist must take to assemble the skin and skeleton. It will also be helpful to know how that data will be exported into X file format. This means we can examine how we load the bone hierarchy and the accompanying skin from

the X file so that we are familiar with all of the data structures and where they come from. This knowledge will make it very easy to understand how to work with skinned meshes in the DirectX pipeline. So let us start our discussions at the art pipeline stage.

While we cannot possibly hope to cover the implementation details of generating skinned meshes in the modeling application (there are entire books devoted to the subject) we will take a look at the basic steps an artist must take in order to create the skinned mesh data for the programmer. This provides extra insight into how the data is generated and stored and thus will aid us in understanding the key concepts of character skinning in our game engine. This section will be kept deliberately brief and generic as the exact process for creating a skinned character and a skeletal hierarchy will vary between modeling applications.

1. Creating the Skin

The first thing the artist will usually do when generating the data for a skinned character system is to build the skin. The skin is just a mesh of the character in its default pose. There is no hard and fast rule about what the default pose should be, but modelers will usually prefer to start off with a pose that allows the skeletal structure to be more easily placed inside the character mesh. As it pertains to building the skin vertex data, this is all the artist typically has to do. Unlike a tweening system where the artist would have to create several copies of the same character mesh in many different poses, in the character skinning system, the artist just has to make this one mesh in its reference pose. Fig 11.8 is an example of a character skin mesh. As you can see, it is just a normal geometric character model and there is nothing special or unique about it. This mesh can be created in whatever modeling application is preferred so long as it can be imported into the application that will be used for building and applying the skeletal structure.

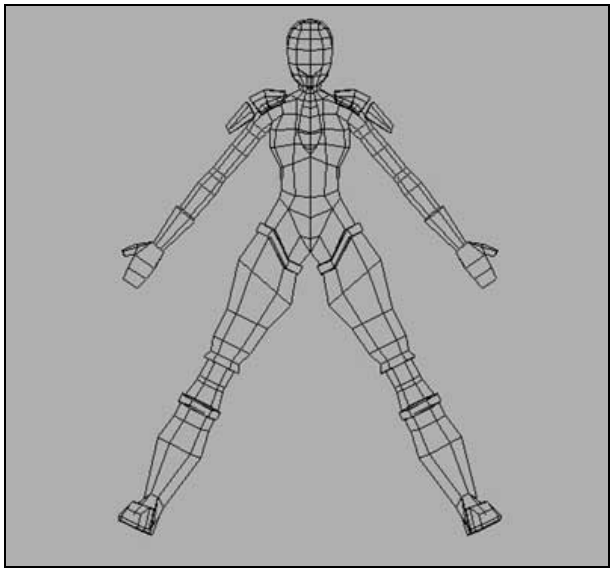


Figure 11.8

Note: The process of creating a set of individual meshes for use in a tweening system can actually be aided by the skeletal animation system most 3D modeling programs provide. For example, an artist can create a skeletal character and animate it within the package, but rather than export the skin and skeleton (which we would do for use in a skinned mesh system), he/she could just save out the transformed meshes at individual keyframes throughout the animation cycle. This essentially means saving the mesh data only and discarding the skeleton data as it will no longer be needed. Whether or not you wish to do this when the skeleton is available depends on the needs of your project, but it is important not to assume that each individual keyframe pose used in tweening must be a unique mesh constructed by hand every time.

2. Applying a skeleton to the skin

Once the mesh is imported into the application that we are going to use to apply the skeletal structure, the next step is for the artist to map this skin to a hierarchy of bones. The process of building and applying a skeleton will vary between 3D modeling packages, but the basic concepts are generally the same. In this discussion we will look at some screenshots of various processes and describe those processes with relation to Character Studio™. Character Studio™ started its life as a powerful animation plug-in tool for 3D Studio MAX™, but in the latest versions of 3D Studio MAX™ has now been fully integrated as part of the MAX package. If you have an earlier version of 3D Studio MAX™ (pre version 5.0) then you will need to purchase the Character Studio™ plug-in separately.

Character Studio™ allows you to visually insert skeletal structures inside of your skin mesh. Although you can build your own skeletal structures (linking one bone at a time together into a hierarchy), Character Studio™ ships with several pre-packaged skeletal structures which you can use directly. The package also ships with numerous test-bed animation scripts (more can be downloaded, purchased or created) that can be assigned directly to the skeletal structure. The test animations run the gamut from smooth motion captured walking scripts to slipping on a banana peel. Many of them are actually quite fun to watch, even using just the default skeleton.

The skeletal structure is referred to as a **Biped** (for obvious reasons) and the animation files are generally stored with a .bip extension. Below you can see the Biped rendered as a simple skeleton in Character Studio™.

While it is not much to look at (perhaps a bit like the Battle Droids from Star Wars™) it is in fact the graphical representation of what will eventually become our bone (frame) hierarchy.

The pre-packaged (or downloadable) test animation scripts are specifically designed to animate this skeletal structure. Therefore, all we have to do is fit this skeletal structure inside our mesh, attach it, and our mesh will benefit from those same animations.

To fit the Biped to the mesh, the artist will scale and orient its limbs to match the reference pose of the skin (this is why it is helpful to keep the arms and legs splayed out away from the torso as shown in Figure 11.8).

Fig 11.9 shows what the current biped might look like after being fitted to the skin shown in Fig 11.8. Notice how the joints of the biped have been rotated and scaled so that they accurately represent what the skeleton of our mesh would look like in its reference pose. The rotation and position of each bone joint will be recorded and will be described by a bone matrix in our hierarchy when the skeletal structure is exported to X file format. We will learn more about that a little later in the lesson.

The mesh with the Biped inside

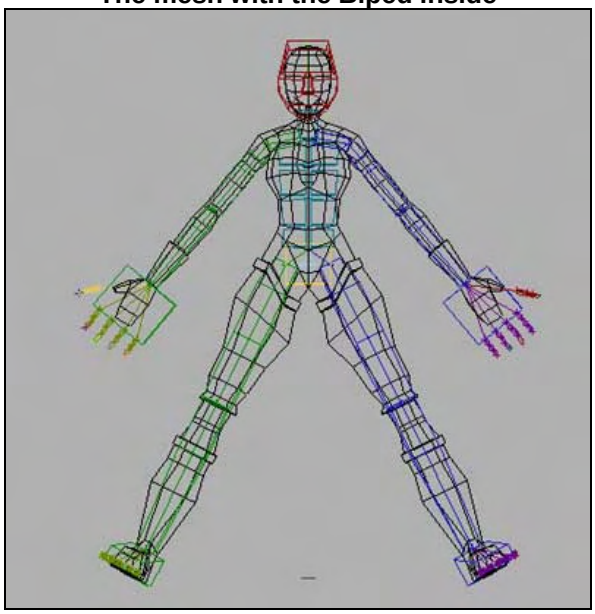


Figure 11.9

Once the skeletal structure has been properly aligned with the skin, the artist will use the 3D Studio MAX™ Physique Modifier to auto-calculate which vertices in the skin are influenced by which bones in the skeleton. It is probable that some (or even many) vertices in the skin will be influenced and thus attached to more than one bone in the skeleton. We saw this earlier with our knee joint.

Furthermore, the weight of each matrix's contribution to each vertex is also calculated for all influential bones for a given vertex. The default calculation that 3D Studio will do for us will involve testing each bone matrix for each vertex to see if that vertex is in a region of influence (also referred to as an *envelope*) for a given matrix. If so, then that vertex will be attached to that bone and the weight of the matrix for that vertex will be calculated as a product of the distance from the vertex to the bone. To calculate the region of influence for a bone, an ellipsoidal bounding volume is used around each bone. Any vertices that fall within this ellipsoidal region will be attached to the bone and influenced by the bone to some degree.

3. Tweaking the bone influences and skin weights

Although the Physique modifier is a completely automated process, after it has been applied, it is not uncommon to find that the mapping of the skin to the skeleton requires some manual tweaking. For example, the default envelope size used to calculate a region of influence for each bone may well

include vertices that we would not like to be influenced by the movement of that bone. Alternatively, we might want the vertex influenced by a bone, but perhaps not to such a large degree. In some cases, we may even find vertices that were not attached to the skeleton at all. By playing the test-bed animation scripts we can easily identify problem areas and correct them. We simply look for places where the skin is not animating in as natural way as we would like given the default matrix contributions and skin weights automatically calculated for us. We will often find that by adjusting the region of influence for a given bone we can address these inaccuracies. The process involves a good deal of trial and error, with us running the animation scripts again and again between tweaks to verify the adjustments just made. Again, this is all in the domain of the project artist, but is helpful to understand how it works (especially if you intend to do your own art and animation).

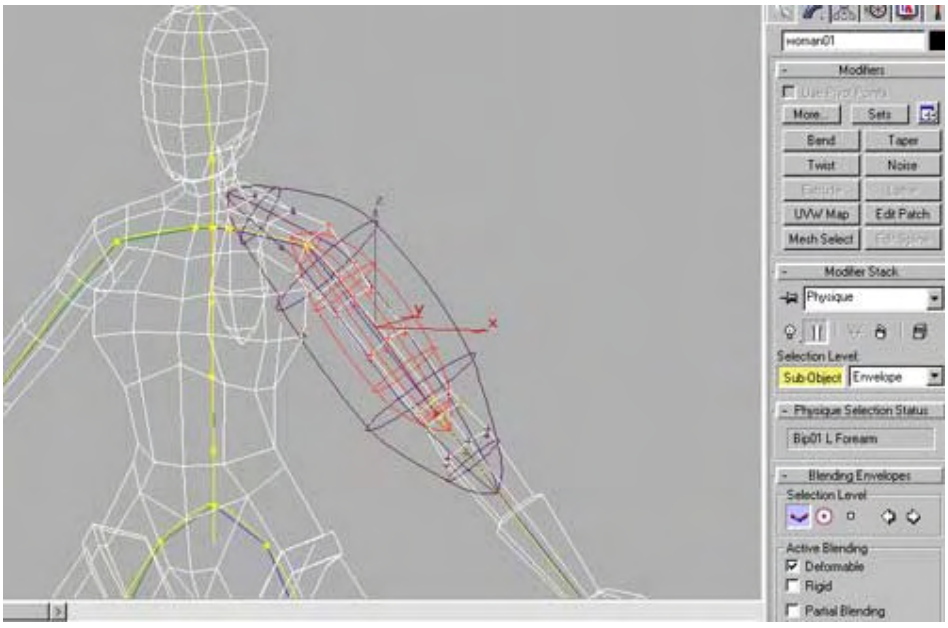


Figure 11.10

Figure 11.10 is a screenshot of an interface for adjusting the envelope (region of influence) for the upper arm bone using the Physique modifier. The size of the ellipsoidal region of influence for that bone can be adjusted to consume more or less of the surrounding vertices of the skin.

4. Applying Animations

Once the skeleton and skin system has been tuned, any number of pre-canned animations can be selected to animate the skeletal structure. This automatically animates the attached skin. There are many animation scripts available for Character Studio™ (usually .bip or .mot files) that can be purchased or downloaded, but most game projects generally entail the artist coming up with original data. Whether this is done by hand in the animation program or by spending time in a motion capture studio, most games involve their own proprietary animation sequences. Ultimately we need keyframes for the animations timeline one way or the other.

5. Exporting the character skin and skeletal structure

Once the artist has the skin and bone system animating correctly, the next step is to get that data into a format that our application can easily load and process. While the DirectX SDK ships with X file exporters for both Maya™ and 3D Studio Max™, there are third party exporters available as well. One of the better free X file exporters available for 3D Studio MAX™ is the Panda™ DX Exporter plug-in. This is a great exporter that converts all the data we need into X file format, including not only the skin and bone data, but the animation data as well. The skeletal structure will be exported as a vanilla X file frame hierarchy and the animation data will be exported as standard keyframe animation data. The only thing we will need to come to grips with is the extra data stored in the X file containing the mapping information which describes which bones in the hierarchy influence which vertices in the mesh. We will discuss this in a moment.

> **Note:** In many file exporters that work with 3D Studio MAX™, you have to be very careful not to mix MAX bones with Character Studio bones (they are different). Keep this in mind when you are designing your skeletons.

If you have 3D Studio MAX™ and wish to use the Panda™ DX Exporter you can visit their website and download it for free at http://www.pandasoft.demon.co.uk/directxmax4.htm.

There are other plug-in exporters available for 3D Studio MAX™ which do not export the data in X file format, but rather in some other easy to read format. The excellent Flexporter™ plug-in, designed by our colleague and friend Pierre Terdiman, exports to a proprietary file format called .ZCB. While .ZCB files are easily read, the real power of Flexporter™ comes from the fact that it is actually a plug-in SDK that lets you define your own file formats and write your own plug-ins. Flexporter™ will simply provide your plug-in with the data it reads from MAX and you can store it however you see fit. This saves you the trouble of having to learn the entire MAX Plug-In SDK. You can download the Flexporter™ plug-in SDK at http://www.codercorner.com/Flexporter.htm.

Finally, we would be remiss if we did not mention some of the non-free export tools available to you as well. One of the most popular workhorse applications available for dealing with export of scene data and animation is Okino Software's PolyTrans™. It is a very powerful tool that supports data import and export from almost all of the popular 3D modeling programs (MAX, Maya, Lightwave, etc.) and it has an X file exporter built in. This means that you can support art and animation from just about every modeling tool on the market and get it all into X format with a few clicks of a button. A similar tool that has most of the same bells and whistles as PolyTrans™ is DeepExploration™. Neither of these tools are free (unlike the other exporters just mentioned), although they are both reasonably priced given their power and flexibility. If you have the budget, either one is a worthwhile investment for any professional project.

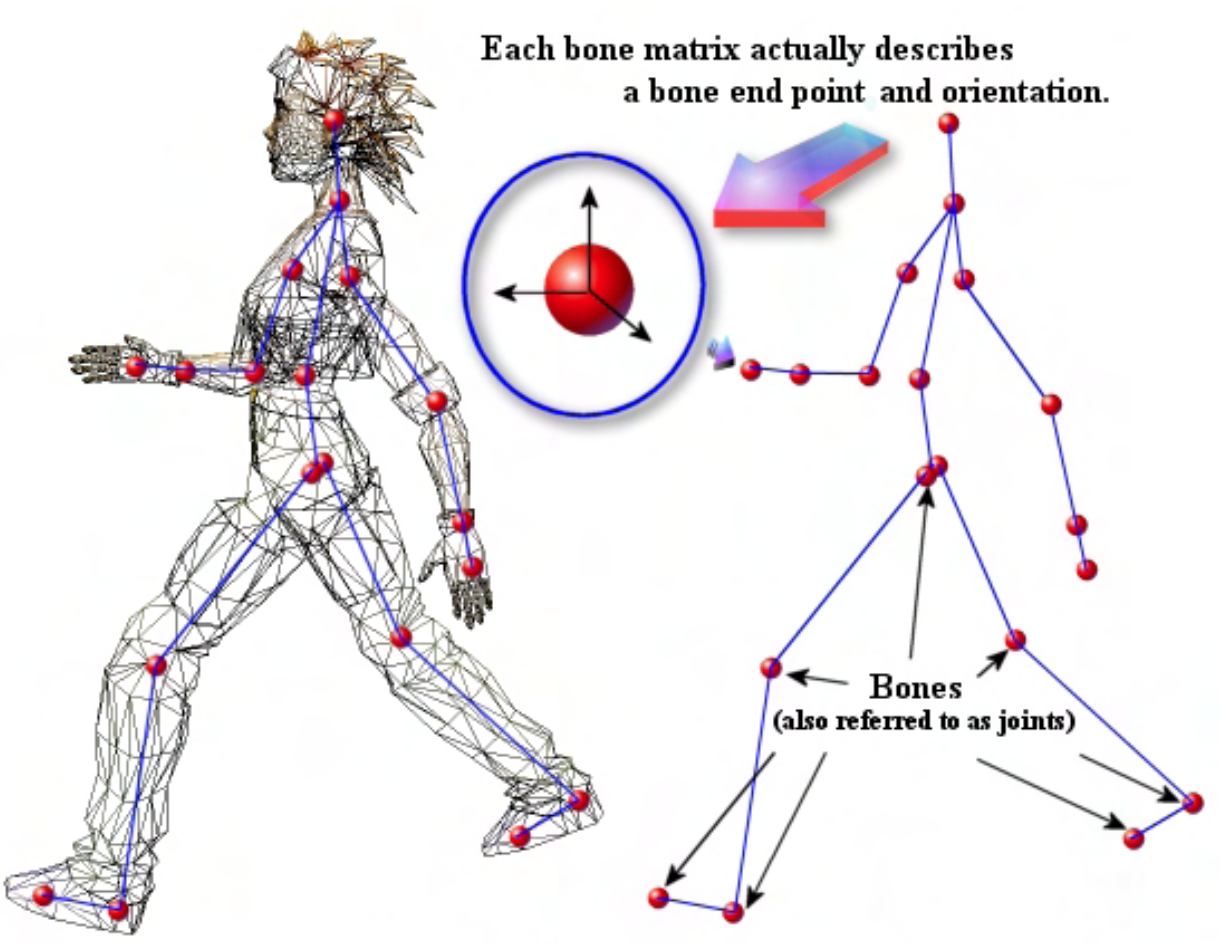
So even if none of the X file exporters fit your needs, you could always use a system like Flexporter™ and write some code to build your own plug-in file format. If you intend to use the D3DX animation system then you will need to write the code that also uses this data to populate the animation controller, its animation sets, and its underlying keyframe interpolators. Fortunately, we learned how to do this in the last chapter, so you have some knowledge of how to approach such a system.

We now understand skin mesh creation at a high level. Although there will be differences between various modeling applications, the underlying system we are creating still consists of two separate entities: the skin and a skeletal structure. When saved out to an X file, the skeleton is saved as a frame hierarchy that D3DXLoadMeshHierarchyFromX will load automatically. Each frame in the X file will be a bone in the skeleton and the matrices stored in each frame will contain the parent-relative transformation of that bone. Each bone in the hierarchy will have also been assigned a name so that the animation controller can manipulate the bone matrix. This also allows us to identify the matrices which affect each vertex. This will all be dealt with at load time and, as we will see later, is something that we will need to know in order to make sure that the Direct3D device has the correct bone matrices available when rendering specific subsets of the mesh.

For the remainder of this chapter we will be assuming that you have exported your skinned character and skeletal animation structure into X file format. You should have a single X file which contains at least one skin, one skeletal hierarchy, and a collection of animation data. You do not have to export keyframe animation data with the skinned character since you may choose to apply the animations to the bone matrices programmatically instead. However, by storing the various animations in separate animations sets inside the X file, you have the benefit of using D3DXLoadMeshHierarchyFromX to load the various animation sets and register them with the animation controller. This way you can just select the pre-canned animation you wish to play, assign it to a track on the animation mixer, and start animating your character immediately. If you do not have a character X file handy to practice with, you can use the samples accompanying this course, or those (like tiny.x) that ship with the SDK.

11.3.2 Bones and Joints

In Fig 11.11 we can see that the animation data manipulates the bones in the skeleton and that because the skin vertices are attached to those bones, the entire mesh is animated as a result. Although in 3D Studio MAX™ (and in Fig 11.11) we have rendered the skeleton as a physical entity existing inside the skin, it will not exist in our applications as a visual object. The bones of the skeleton are ultimately just matrices describing a position and orientation relative to a parent bone in the hierarchy. We render them just to get a feel for what is happening behind the scenes.



It is common to represent the hierarchy in diagrams by connecting up the bone matrix positions with lines to form a skeleton. Whilst this is useful for visualizing the underlying skeleton that the mesh is using, we must remember that in such diagrams the lines themselves are not what our bone matrices represent, each bone matrix actually represents a bone end point.

Figure 11.11

As Fig 11.11 mentions, it is very common for people to think of the bones themselves as being the links between joints. While of course this is technically true in real life, it is important to bear in mind that a bone in our hierarchy behaves much more like a joint. That is, In the context of a skeletal structure in computer graphics, it is perhaps more intuitive to think of a bone in the hierarchy as being just the bone end-point (or a ***joint***).

Recall that the 'bone' is a frame of reference in our system. Therefore, in the diagram we can see that the bones in our hierarchy will actually be the red spheres shown in the skeleton and technically not the blue lines connecting them. It can be a bit confusing that the word 'bone' is the more common way to refer to these frames instead of the (perhaps more applicable) term 'joint' which better describes what is being animated. Technically, it is really both (the joint plus the bone that emerges from it – much like the Z axis emerging from a coordinate system origin) but the actions that we apply to a given bone really take place at the joint itself (i.e. the system origin).

> **Note:** Many modeling applications and texts on the subject of 3D skeletal animation still refer to bones as 'joints'. We will use the 'bone' terminology in this course in keeping with the DirectX naming convention for such entities but just try to keep this concept in the back of your mind.

To really understand the bone/joint equivalence concept, take another look at Fig 11.3. Because we are describing Frame 1 in this example as being the 'upper right leg bone', we might envisage this bone as being a long white thing that spans the length of the upper leg mesh. While this is understandable, and not entirely false, just keep in mind that a bone is described by a matrix that contains a position and an orientation. The important point to note is that the position in that frame matrix describes the *end-point* of the bone that starts at the top of the leg. Keep in mind that like any line, the term 'end-point' can apply to both the terminal point (the 'end') of the bone as well as its origin (the 'beginning'), but that we are referring to the origin. The bone extends down through the length of the thigh to the kneecap (the next bone in the hierarchy), but rotations are governed by what takes place at the origin (the hip).

In this discussion we have changed nothing from the previous chapter; we have simply started referring to things using different names. Because the frame hierarchy is being used to represent a character, we are referring to it as a skeleton, with each frame as a bone and each frame matrix as a bone matrix. The bone matrix still describes the position and orientation of the body part mesh that is attached to that bone relative to the parent bone.

In the next section we will examine how the skinned mesh(es) will be stored in an X file. This will prove to be a very short discussion since most of the standard templates we have already covered in the previous chapter will be used again. While it is not strictly necessary for you to understand how the skinned mesh and skeletal structures are stored in an X file in order to be able to load and render them, examining how the data is stored (especially the way that the vertices are mapped to bones in the hierarchy along with weights) will help greatly in understanding how to store and render that data once it is loaded.

11.4 X File Skins and Skeletons

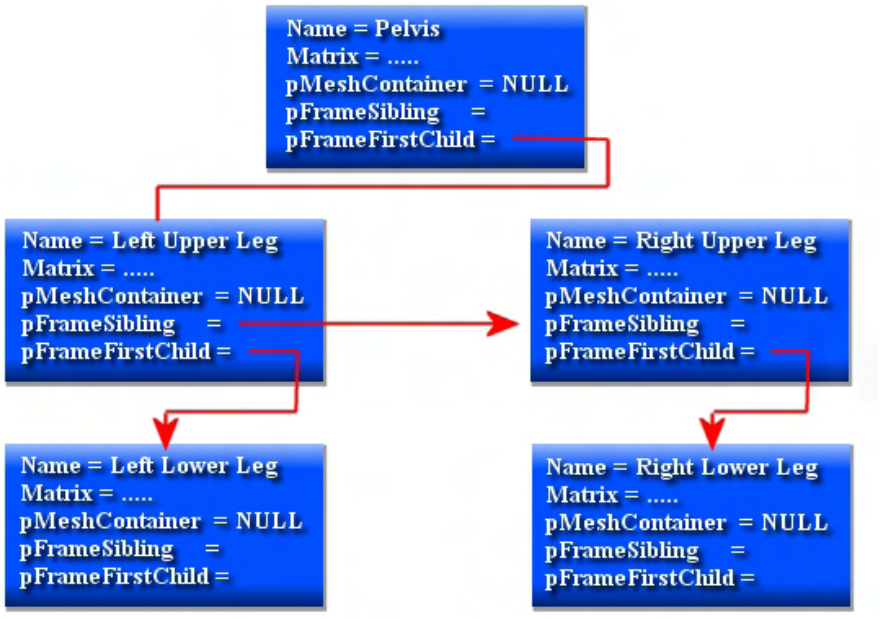
When a skinned mesh and a skeletal structure are exported to an X file, the skeletal structure is exported as a generic frame hierarchy, where each frame is a bone. The name of the frame will be the name that the artist assigned to the bone in the modeling package. This frame hierarchy is represented using the standard Frame template that we looked at earlier in the course. If we use our example of a pair of legs constructed from a skeletal structure consisting of five bones (a pelvis, two upper leg bones and two lower leg bones) we can see that the skeleton is represented in the X file as shown in the file snippet that follows.

Recall that each frame contains a transformation matrix (which we will now refer to as the parent-relative bone matrix) and any number of child frames. As we can see in this example, the pelvis frame is the root frame, which contains two immediate child frames -- the left upper leg bone and the right upper leg bone. Each one of these two child bones themselves have a child bone -- the left upper leg bone has the left lower leg bone as a child and the right upper leg bone has the right lower leg bone as a child. Notice that each bone in the skeletal structure that will be animated and have vertices assigned to it is given a name. In this example we have just set the matrix data of each bone to identity, but in a real world situation this would contain the position and rotation information of the bone.

```
Frame Pelvis   // Root Frame
{
      FrameTransformMatrix  // Pelvis Bone Matrix
      {
       1.000000,0.000000,0.000000,0.000000,
       0.000000,1.000000,0.000000,0.000000,
       0.000000,0.000000,1.000000,0.000000,
       0.000000,0.000000,0.000000,1.000000;;
      }

      Frame LeftUpperLeg            // Child Bone of Pelvis
      {
         FrameTransformMatrix       // LeftUpperLeg Bone Matrix
         {
           1.000000,0.000000,0.000000,0.000000,
           0.000000,1.000000,0.000000,0.000000,
           0.000000,0.000000,1.000000,0.000000,
           0.000000,0.000000,0.000000,1.000000;;
         }

      Frame LeftLowerLeg      // Child of LeftUpperLeg Bone
      {
        FrameTransformMatrix // LeftLowerLeg Bone Matrix
        {
          1.000000,0.000000,0.000000,0.000000,
          0.000000,1.000000,0.000000,0.000000,
          0.000000,0.000000,1.000000,0.000000,
          0.000000,0.000000,0.000000,1.000000;;
        }
      }
   }
```

```
  Frame RightUpperLeg         // Child Bone of Pelvis
  {
      FrameTransformMatrix          // RightUpperLeg Bone Matrix
      {
       1.000000,0.000000,0.000000,0.000000,
       0.000000,1.000000,0.000000,0.000000,
       0.000000,0.000000,1.000000,0.000000,
       0.000000,0.000000,0.000000,1.000000;;
      }

      Frame RightLowerLeg           // Child Bone of RightUpperLeg
      {
         FrameTransformMatrix       // RightLowerLeg Bone Matrix
         {
          1.000000,0.000000,0.000000,0.000000,
          0.000000,1.000000,0.000000,0.000000,
          0.000000,0.000000,1.000000,0.000000,
          0.000000,0.000000,0.000000,1.000000;;
         }
      }
   }
}
```

Forgetting about how the character skin is stored for now, we know that if we were to use the D3DXLoadMeshHierarchyFromX function to load this hierarchy it would create a D3DXFrame hierarchy for our skeleton as shown in Fig 11.12.



The D3DXFrame Hierarchy that would be created by the D3DXLoadMeshHierarchyFromX Function

Figure 11.12

Our application would be returned a pointer to the root frame (the pelvis bone). The pelvis bone would have its child pointer pointing at the LeftUpperLeg bone, which would be connected to the RightUpperLeg bone via its sibling pointer. So, both of these frames are considered to be child bones of the pelvis bone. The LeftUpperLeft bone would have its child pointer pointing at the LeftLowerLeg bone and the RightUpperLeg bone would have its child pointer pointing at the RightLowerLeg bone.

However, while the frame hierarchy representation in the X file is unchanged when representing bones, the big change in a skin and bones representation is that there is no longer a mesh attached to the various frames in the hierarchy. So, where exactly is this single skin (mesh) data stored?

As it turns out, there is no hard and fast rule about where the skin needs to be stored. What we do know is that it will no longer be assigned to a single frame in the hierarchy and that it will have the appropriate information contained inside the Mesh data object describing which vertices map to which bones. Therefore, the mesh might be embedded as a child mesh of the root frame or even some arbitrary child frame. Because the position of the mesh in the hierarchy no longer directly dictates which frames animate it (this will be described by a bone table inside the mesh object) or the order in which rendering takes place, the skin can pretty much be defined anywhere in the hierarchy (although it is generally defined in the root frame). When D3DXLoadMeshHierarchyFromX detects that the Mesh data object has 'skinning information' inside its data block, it knows to treat this mesh differently and supply our application with bone-to-vertex mapping information. Note that we no longer need to render the mesh hierarchically because it is a single mesh (although we still need to update the hierarchy matrices recursively). So if the Mesh data object is defined inside the root frame, then the root frame's pMeshContainer will point to the mesh container that stores the skin. This is the pointer we will use to access and render the subsets of the mesh.

You will be pleased to know that the definition of a skin mesh is no different from the regular Mesh X file definition. All of the same standard templates are used to define the Mesh data object itself as well as the child data objects such as the material list, the texture coordinates, and vertex normals. The only difference with a mesh that will be used for skinning is that it will contain additional child data objects. So we might imagine an arrangement as shown in the next example, where the Mesh object is defined as a child of the root frame. In this example we have not shown ***all*** of the mesh data and child objects, but you will get the general idea.

```
Frame Pelvis   // Root Frame
{
      FrameTransformMatrix  // Pelvis Bone Matrix
      {
        1.000000,0.000000,0.000000,0.000000,
        0.000000,1.000000,0.000000,0.000000,
        0.000000,0.000000,1.000000,0.000000,
        0.000000,0.000000,0.000000,1.000000;;
      }

      Mesh LegsSkin
      {
            4432;                         // Number Of Vertices
            -34.720058;-12.484819;48.088928;,   // Vertex List
            -25.565304;-9.924385;26.239328;,
            -34.612186;-1.674418;34.789925;,
```

```
             0.141491;7.622670;25.743210;,
            -34.612175;17.843525;39.827816;,
            -9.608727;27.597115;38.148296;,
            …  Remaining Vertex Data Goes Here….

            6841;              // Number of Faces
            3;28,62,1;, // Face List
            3;3,16,3420;,
            3;11,23,29;,
            3;104,69,7;,
            3;0,13,70;,
            3;9,97,96;
            … Remaining Face Definitions Go Here …


            MeshNormals
            {
                  4432;
                  -0.989571;-0.011953;-0.143551;,
                  -0.433214;-0.193876;-0.880192;,
                  -0.984781;0.061328;-0.162622;,
                  -0.000005;0.123093;-0.992395;,
                  … Remaining Mesh Normals List Goes Here …..
            }

            .. Other child objects go here, Texture coordinates
               and a Mesh Materials List for example…

       }  // End mesh definition

       Frame LeftUpperLeg           // Child Bone of Pelvis
       {
            FrameTransformMatrix    // LeftUpperLeg Bone Matrix
            {
              1.000000,0.000000,0.000000,0.000000,
              0.000000,1.000000,0.000000,0.000000,
              0.000000,0.000000,1.000000,0.000000,
              0.000000,0.000000,0.000000,1.000000;;
            }

            Frame LeftLowerLeg            // Child of LeftUpperLeg Bone
            {
                  FrameTransformMatrix    // LeftLowerLeg Bone Matrix
                  {
                     1.000000,0.000000,0.000000,0.000000,
                     0.000000,1.000000,0.000000,0.000000,
                     0.000000,0.000000,1.000000,0.000000,
                     0.000000,0.000000,0.000000,1.000000;;
                  }
            }
      }

      Frame RightUpperLeg           // Child Bone of Pelvis
      {
            FrameTransformMatrix          // RightUpperLeg Bone Matrix
            {
```

```
                1.000000,0.000000,0.000000,0.000000,
                0.000000,1.000000,0.000000,0.000000,
                0.000000,0.000000,1.000000,0.000000,
                0.000000,0.000000,0.000000,1.000000;;
            }

            Frame RightLowerLeg           // Child Bone of RightUpperLeg
            {
                  FrameTransformMatrix    // RightLowerLeg Bone Matrix
                  {
                     1.000000,0.000000,0.000000,0.000000,
                     0.000000,1.000000,0.000000,0.000000,
                     0.000000,0.000000,1.000000,0.000000,
                     0.000000,0.000000,0.000000,1.000000;;
                  }
             }
      }
}
```

Fig 11.13 shows how the loaded hierarchy and mesh container would be arranged in memory after the D3DXLoadMeshHierarchyFromX function has finished executing. There is a single mesh attached to the root frame although, as mentioned, the mesh could have been attached to another frame in the hierarchy. Your application will probably want to store this mesh pointer in some other more easily accessible application variable. Fetching from the root frame structure every time is also fine.

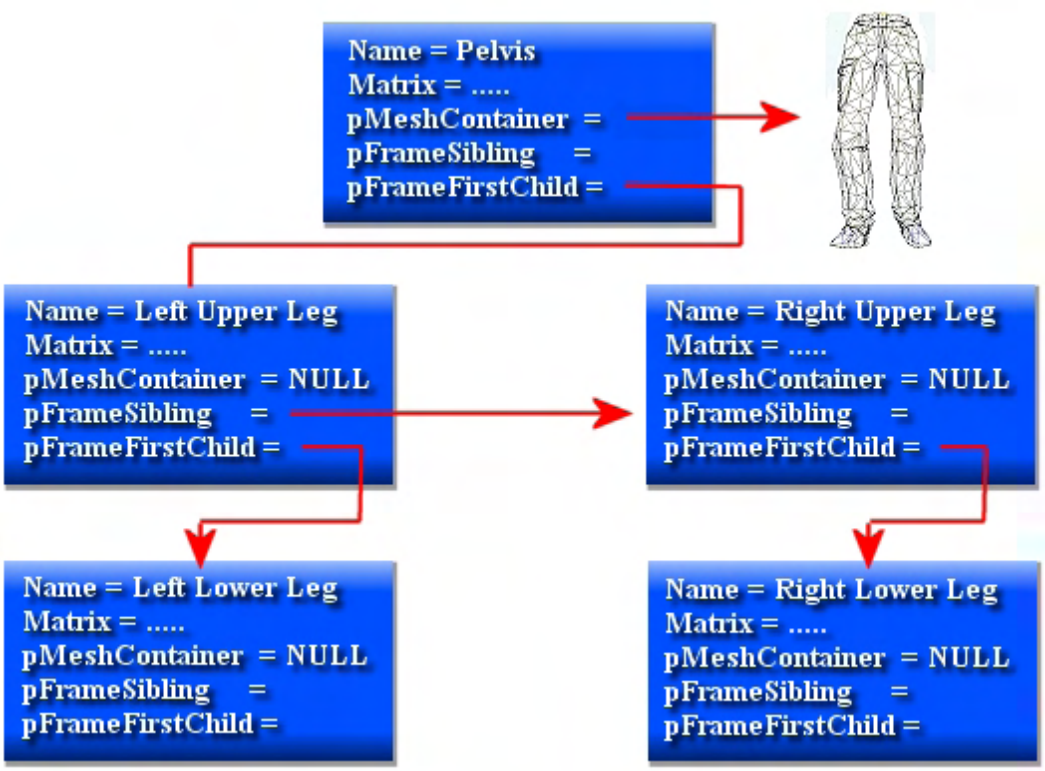


Figure 11.13

While things have not changed much from our previous chapters, we still need to address the bone/vertex relationship. As it happens, a skin mesh data object will be like a normal mesh data object with the exception that it will now have two additional types of child data objects embedded within its definition (not shown in Fig 11.13). These new object types are both standard X file templates. Together they will contain the information that describes which vertices a bone affects and by how much. The existence of these standard templates in the file is how D3DXLoadMeshHierarchyFromX identifies the mesh as a skin and thus takes additional action to load the bone-to-vertex mapping table. We will examine these templates in the next two sections.

11.4.1 The XSkinMeshHeader Template

The first standard template we will discuss is the XSkinMeshHeader template. There will be exactly one of these data objects defined as a child of the mesh data object. This template stores information about the maximum number of bone matrices that affect a single vertex in the mesh and the maximum number of bone matrices that are needed to render a given face. A 'face' is a triangle consisting of three vertices, so if (for example) each of a triangle's vertices were influenced by four unique bone matrices, then we know that in order to transform and render this triangle we would need to have access to the (3 * 4) 12 matrices that influence this single face.

It may not yet be obvious why we would need to know how many matrices collectively affect a face. After all, we are just transforming each vertex into world space, one at a time, using the matrices that affect that vertex. It would seem that all we would need to know is how many matrices affect the specific vertex being transformed. As it turns out, this is true when we perform software skinning (discussed later), where we basically just perform vertex blending one vertex at a time. However, if the hardware supports vertex blending, then things will change a bit. We learned early in this course series that when calling the DrawPrimitive family of functions, the pipeline transforms our vertices in triangle order (i.e. we pass in lists of triangles to be rendered and they are processed one at a time). Thus, in order for the pipeline to transform a triangle's vertices using vertex blending, the device must have access to all of the matrices that influence all of the triangle vertices before we issue the DrawPrimitive call. All of this will make a lot more sense later on in this chapter when we start looking at API specific skinning.

Let us first have a look at the XSkinMeshHeader type. It is a very simple template with just three members. Again, there will be exactly one of these child data objects inside each skinned mesh data object.

```
template XSkinMeshHeader
{
     <3cf169ce-ff7c-44ab-93c0-f78f62d172e2>
     WORD nMaxSkinWeightsPerVertex;
     WORD nMaxSkinWeightsPerFace;
     WORD nBones;
}
```

WORD nMaxSkinWeightsPerVertex
This will describe the maximum number of bones (matrices) that influence a vertex in the mesh. For example, some vertices might only be influenced by a single bone, others by two. Perhaps there are also a few vertices that are affected by three bone matrices. In this example, this value would be set to 3 to let DirectX know that at most, a vertex in this mesh will only be influenced by three matrices. This is actually very useful information. We will see later that when we create a skinned mesh for hardware skinning, the vertex format used will also contain a number of floating point weight values per vertex. These weights describe the influence for each bone that affects it. Since all of the vertices in a mesh's vertex buffer will need to be the same format, we need to make sure we specify the correct FVF flags when creating the mesh. We want to ensure that enough storage space is allocated per-vertex for the maximum number of weights needed by any given vertex in the mesh.

WORD nMaxSkinWeights
This value will describe the maximum number of bones that influence a single triangle in the mesh. If a triangle in the mesh used four unique bones for ***each of its vertices*** but other triangles in the mesh used fewer, this value would still be set to 4*3 = 12. This will tell our application how many matrices we must make available (at most) to the device in order for a given triangle to be transformed by the pipeline using vertex blending. Not all triangles will need access to this many matrices to be transformed and rendered, but the system must be setup such that we can cope with those that do.

WORD nBones
This value describes how many bones (frames) in the hierarchy influence this skinned mesh. At first you might assume that all bones must affect the mesh, but that would not be correct. This is because the X file might contain multiple skeletal structures all linked in one big hierarchy and it might also contain multiple skins for those skeletons. In that case we could say that the hierarchy contains multiple skinned meshes where each mesh is mapped only to the bones for its own skeleton in the hierarchy. The X file might also contain an entire scene hierarchy of which a skinned mesh's bone hierarchy is just a subset. Suffice to say, all of the frames defined in the X file might not belong to a single skeletal structure or might not belong to the skeletal structure attached to a given skin. Quite often however, a skinned mesh and its skeletal structure will be stored in its own separate X file.

11.4.2 The SkinWeights Template

The SkinWeights template stores information about which vertices in the parent mesh data object are affected by a single bone in the hierarchy. It also stores the weights by which those vertices are influenced by the bone. Therefore, the nBones member of the XSkinMeshHeader data object describes how many SkinWeights child data objects will follow it in the mesh definition. If a skin's skeleton consists of five bones, there will be one XSkinMeshHeader child data object followed by five SkinWeights data objects (one SkinWeights child data object defined for each bone in the skeleton for this mesh).

```
template SkinWeights
{
      <6f0d123b-bad2-4167-a0d0-80224f25fabb>
      STRING transformNodeName;
      DWORD nWeights;
      array DWORD vertexIndices[nWeights];
      array FLOAT weights[nWeights];
      Matrix4x4 matrixOffset;
}
```

STRING transformNodeName

This member is a string that will store the name of the bone for which this information will apply. For example, if this string contained the name RightUpperLeg, then the data that follows (the vertices attached to this bone and their weights) will describe the vertices attached to the RightUpperLeg bone in the skeletal hierarchy.

DWORD nWeights

This value describes how many vertices in the skin will be influenced by this bone. As a result, it will tell us the size of the two arrays that follow this member which contain vertex indices and vertex weights for this bone.

array DWORD vertexIndices [nWeights]

Following the nWeights member is an array of vertex indices. The number of vertices in this array is given by the nWeights member. For example, if nWeights = 5 and the vertexIndices array held the five values {4, 7, 20, 45, 100} then it means that the fourth, seventh, twentieth, forty-fifth and one hundredth vertex defined in the parent mesh's vertex list are mapped to this bone. Therefore, when these vertices are being transformed into world space, this bone's matrix should be used to some extent. We deduce then that if a vertex is influenced by *n* bones, then its index will be included in the vertexIndices array of *n* SkinWeights data objects.

array DWORD weights[nWeights]

We looked at some pseudo-code earlier in the chapter that performed vertex blending using multiple matrices. Recall that we multiplied the model space vertex with each bone matrix (one at a time) to create *n* temporary vertex positions (where *n* was the number of matrices being used in the blend). Each temporary vertex position was then scaled by the weight assigned to the matrix that transformed it, which scaled the contribution of the matrix. Recall as well that the combined weights are equal to 1.0 and that the sum of these scaled temporary vectors was the final vertex position in world space. So we know that it is not enough to define for each bone the vertices that are influenced by it; we must also define weights describing the strength of the bone's influence on each of those vertices. Therefore, following the vertexIndices array is another array of equal size defining the weight of this bone on the corresponding vertex in the vertexIndices array. Each of these weights will usually be a value between 0.0 and 1.0.

Matrix4x4 matrixOffset

The final member of the SkinWeights data object is the bone offset matrix. The bone offset matrix is often a cause for much confusion for students who are new to skeletal animation. We will examine the bone offset matrix in detail very shortly.

Before we finish our X file discussion, let us look at an example of how the skin/bone data is stored in the file. To save space, we will not show our leg hierarchy again, we will just concentrate on the mesh data object (which we already know will exist as a child of one of the frame data objects). We will also pare down the amount of mesh data and standard child objects shown so that we can concentrate on the skin information. We really just want to focus on how the XSkinHeader data object and the multiple SkinWeights data objects will be embedded as child data objects of the mesh. We will not include the SkinWeights data object for each bone (to save space) and instead will simply show two of them along with their vertex mappings. Since our skeleton actually consists of five bones, we know that in reality there would be five SkinWeights data objects defined for this mesh.

```
Mesh LegsSkin
{
      300;                          // Number Of Vertices
      -34.720058;-12.484819;48.088928;,   // Vertex List
      -25.565304;-9.924385;26.239328;,
      -34.612186;-1.674418;34.789925;,
       0.141491;7.622670;25.743210;,
      -34.612175;17.843525;39.827816;,
      -9.608727;27.597115;38.148296;,
      … Remaining Vertex Data Goes Here….

      200;        // Number of Faces
      3;28,62,1;, // Face List
      3;3,16,3420;,
      3;11,23,29;,
      3;104,69,7;,
      3;0,13,70;,
      3;9,97,96;
      … Remaining Face Definitions Go Here …

      MeshNormals
      {
            4432;
            -0.989571;-0.011953;-0.143551;,
            -0.433214;-0.193876;-0.880192;,
            -0.984781;0.061328;-0.162622;,
            -0.000005;0.123093;-0.992395;,
            … Remaining Mesh Normals List Goes Here …..
      }
      XSkinMeshHeader
      {
        2;     // Each vertex is assigned to no more than two bones
        4;     // The maximum bones used by a single triangle is 4
        5;     // This skin uses a five bone skeleton
      }

      SkinWeights     // Mapping information for the ‘RightUpperLeg’ bone.
      {
            "RightUpperLeg";
            2;                // Two vertices are mapped to this bone.
            150,              // Vertex 150 is one of them
            300;              // And vertex 300 is the other

            0.500000,         // Vertex 150 has a weight of 0.5 for this bone
```

```
            0.500000;         // Vertex 300 also has a weight of 0.5 for this

            //bone offset matrix for the RightUpperLeg bone
            1.253361,   -0.000002,   0.254069,   0.000000,
           -0.218659,   0.223923,    1.078679,   0.000000,
            0.058231,   -1.440720,   -0.287275,  0.000000,
           -8.131670,   62.204407,   -2.611076,  1.000000;;
      }

      SkinWeights   // Mapping information for the 'LeftUpperLeg' bone.
      {
            "LeftUpperLeg";
            3;                // Three vertices mapped to this bone
            1,                // Vertex 1 is mapped to this bone
            20,               // Vertex 2 is another
            25;               // Vertex 3 is another

            0.333333,         // Vertex 1 weight for this bone
            0.333333,         // Vertex 2 weight for this bone
            0.333333;         // Vertex 3 weight for this bone

            // Bone offset matrix for left upper leg bone
            1.253361,   -0.000002,    0.254069,  0.000000,
           -0.218659,    0.223923,    1.078679,  0.000000,
            0.058231,   -1.440720,   -0.287275,  0.000000,
           -8.131670,   62.204407,   -2.611076,  1.000000;;
      }

… Remaining SkinWeights objects defined here. There should be 5 in total, one for each bone…
}  // End mesh definition
```

So we can see that the data is stored in the X file in a simple fashion. There will be a hierarchy of bones and a single mesh. That mesh will contain SkinWeights data objects (one for each bone) describing which vertices in the mesh are attached to each bone in the hierarchy, along with the weight describing the influence of that bone matrix on its attached vertices. Furthermore, each SkinWeights data object also contains a bone offset matrix for the bone being described. Before we talk about loading the hierarchy and configuring the API for skinned mesh rendering, let us first examine the bone offset matrix.

11.5 The Bone Offset Matrix

The bone offset matrix stores the inverse transformation of the bone's absolute (not relative) position and orientation in the hierarchy with the skeleton in its reference pose. Although bone offset matrices are provided for each bone in the X file, you could create you own bone offset matrix for each frame by traversing the hierarchy (arranged in its reference pose) starting at the root, and for each frame calculate the absolute matrix for that frame (remember that they are stored as relative matrices in the hierarchy initially). Once you have the absolute matrix for a given frame, just invert it and you will have the bone offset matrix for that bone. The following snippet of code should make this clear.

Note: The code sample that follows is for illustration only. This information is already contained in the X file and loaded automatically by the D3DXLoadMeshHiearchyFromX function.

In the next code example, we have derived our own structure from D3DXFrame just as we did in the last chapter. Recall that we added an additional member called the combined matrix (i.e. the world space matrix of the frame). We will do the same thing again this time, but also add a new member to store the bone offset matrix that we will generate.

Just as a reminder, the standard D3DXFRAME structure is defined as:

```
typedef struct _D3DXFRAME
{
      LPTSTR Name;
      D3DXMATRIX TransformationMatrix;
      LPD3DXMESHCONTAINER pMeshContainer;
      struct _D3DXFRAME *pFrameSibling;
      struct _D3DXFRAME *pFrameFirstChild;
} D3DXFRAME, *LPD3DXFRAME;
```

We will derive our own structure from it with one matrix to contain the absolute transform for each frame when we update the hierarchy and another to contain the bone offset matrix for the given bone.

```
struct D3DXFRAME_DERIVED : public D3DXFRAME
{
      D3DXMATRIX mtxCombined;
      D3DXMATRIX mtxBoneOffset;
}
```

The following function would then be called when the bone hierarchy is first constructed and is still in its reference pose (i.e. before any animations have been applied). ***pRoot*** is assumed to be the root bone returned to the application by D3DXLoadMeshHierarchyFromX. Thus it would be called as:

```
CalculateBoneOffsetMatrices( pRoot, NULL );
```

The function implementation is almost identical to the UpdateFrameHierarchy function we wrote in the previous chapters. It steps through the bone hierarchy using recursion, combining relative matrices along the way to calculate the absolute matrix for each frame. We can think of the combined matrix as being the absolute bone matrix. It describes the actual position of that bone in the representation. Once we have the bone matrix, we invert it to get the bone offset matrix. This offset matrix describes an inverse transformation that provides access to the local space of the bone (as it was defined in its reference pose). You should refer back to our discussion in Chapter Four about the relationship between local space and inverse transformations if this does not sound familiar to you.

```
void CalculateBoneOffsetMatrices (D3DXFrame *pFrame , D3DXMATRIX *pParentMatrix)
{
    D3DXFRAME_BONE * pMtxFrame = (D3DXFRAME_BONE*) pFrame;

    if( pParentMatrix != NULL)
        D3DXMatrixMultiply( &pMtxFrame->mtxCombined,
                            &pMtxFrame->TransformationMatrix, pParentMatrix);
```

```
    else
        pMtxFrame->mtxCombined = pMtxFrame->TransformationMatrix;

    // Calculate the bone offset matrix for this frame
    D3DXMatrixInverse (pMtxBoneOffset, NULL, pMtxCombined);

    if(pMtxFrame->pFrameSibling)
      UpdateFrameMatrices( pMtxFrame->pFrameSibling, pParentMatrix );
    if(pMtxFrame->pFrameFirstChild)
      UpdateFrameMatrices(pMtxFrame->pFrameFirstChild,
                          &pMtxFrame->mtxCombined );
}
```

It is very important to note that while the bone matrices in the hierarchy change when animations are applied to them, this is not true of the bone offset matrices. They will not change throughout the life of the character (otherwise there would be little point storing the information in the X file). Using code similar to the above example, they would be calculated once when the mesh is first loaded and stored away for later use. Remember, we never need to calculate them as shown above; this code was provided as a means of describing how the bone matrices stored in the X file were initially calculated by the modeling application that created it.

11.5.1 The Necessity of the Bone Offset Matrix

Early on in this chapter we looked at some vertex blending code that showed a vertex in the skin being transformed from model space to world space using a blend of multiple world matrices. If we were to use this function to transform the vertices of our skin, the matrices we would use to transform a given vertex would be the absolute bone matrices calculated when we updated the frame hierarchy. We would certainly not apply the relative transformations because we wish to render the vertex and thus require that it exist in world space. So before we transform the skin vertices from model space to world space, we would first need to apply any relevant animations to the frame hierarchy (using the ID3DXAnimationController::AdvanceTime) method. These animations result in a new set of parent-relative frame matrices being stored (as described in the previous chapter). Then we would traverse the hierarchy and concatenate all of the appropriate matrices together (as we did in the previous chapter also) to update the absolute matrices stored in each frame. These combined matrices in each bone store the world space transform for the bone and thus should be used to transform any attached vertices during vertex blending.

Without the bone offset matrix, this final transformation presents a problem. Imagine for the sake of simplicity that we are transforming a single vertex in the skin from model space to world space. Also imagine that the vertex is only attached to one bone (e.g., RightUpperLeg) and that the vertex has a weight of 1.0 stored for that bone. We might assume that we could transform the vertex into world space by doing the following (given that mtxCombined has just been updated to contain the correct position and orientation of the bone in world space):

World Space Vertex = Model Space Vertex * RightUpperLeg->mtxCombined.

This looks like it should work, until we remember that the vertex is defined in model space. That is, the vertex is defined relative to the origin of the model space coordinate system instead of being defined relative to the bone via which it is being transformed. Why is this necessary? Well let us think for a moment about what we would like to have happen. If an animation is applied to a bone, we want that motion to propagate out to all of the skin vertices that are attached to that bone so that they move along with it. Where have we seen this sort of behavior before? In Chapter Ten, of course. When we animated a given frame in our hierarchy, that animation was propagated down to all of its children and they experienced the same animation themselves. This is exactly what we want to have happen in our bone/vertex case. But recall that our frame hierarchy is set up in a relative fashion. That is, each frame is defined in the local space of its parent. Unfortunately this is not the case for our vertices. We know that the vertices of our skin are defined relative to the origin of the model (their 'parent', if you will). They exist in model space. So if we want them to move along with the bone, they will need to be defined relative to the bone (i.e. exist in bone local space). Or more to the point, we want to attach our vertices to the hierarchy such that they behave as children of the bone.

So then why not just store the vertex as a position relative to the bone instead of the model space origin? That is not possible either because the vertex may need to be transformed by multiple bones, and in each case the vertex would need to be defined relative to that bone. That is clearly not going to work unless we wish to store multiple copies of our vertices (which is problematic and obviously not very appealing).

Just for clarity, let us see what happens anyway if we simply transform the vertex by the bone matrix. To simplify this example we will forget about orientation for now and just concentrate on the position of a bone in its reference pose.

Let us imagine that we have a bone that has a combined matrix such that, in its reference pose it is at position **b(50,50,50)** . Let us also assume that we have a vertex that will be influenced by this bone and it is positioned at position **v(60,60,60)** with respect to the model space origin (in its reference pose). We can see right away that the vertex is offset (10,10,10) from the bone when both the bone hierarchy and the skin are in the reference pose. So what we are going to do next is simply assume that the skeleton is still in its reference pose and transform our vertex by our bone matrix.

We know for starters that the vertex should be at position (60,60,60) in its default pose. So if we used the above transformation approach, this is where the vertex should be after being multiplied by the bone matrix (since the bone has also not moved from the default pose). You might see already that this is clearly not going to work. If we think about the case of simply transforming and rendering the mesh in its reference pose, the vertex is already in the correct position (with respect to the bone) even before we transform it using the bone matrix. If we were to transform vertex **v** by bone **b**, we know that with the bone at (50,50,50) the vertex should be at (60,60,60). But look at what happens next:

$$\begin{aligned} \textit{World Space Vertex} &= v * b \\ &= (60, 60, 60) + (50, 50, 50) \\ &= (110, 110, 110) \end{aligned}$$

Note: We are dealing with only the translation portion of the bone matrix to make the example easier. We are assuming that the bone is not rotated in its default pose. Therefore **v** * **b** = **v** + **b**.

Of course, if the vertex was originally defined in bone space instead of model space then the position of the vertex would have been stored as v(10,10,10). This vertex position which is defined relative to the position of its influential bone would indeed create a transformed vertex position of (60,60,60) as expected.

So given that our vertices are not defined in bone space, but in model space, what we need to do before we transform any vertex by a given bone matrix is first transform the vertex into that bone's local space. This way, the vertex is defined in bone space *prior to* being transformed by the bone matrix. Using our simple example again, this would mean performing an inverse translation of the bone's position for the vertex to convert it from a model space vertex position to a bone space vertex position.

First we transform model space vertex **v** into the space of bone **b** by subtracting the bone's offset (i.e. position) from the model space vertex.

Bone Space Vertex = v * -b = (60 , 60 , 60) + (-50 , -50 , -50)
= (10 , 10 , 10)

We now have the vertex in bone space, so we can just apply the combined bone matrix to transform the bone space vertex into world space:

World Space Vertex = Bone Space Vertex * b = (10 ,10 , 10) + (50 , 50 , 50)
= (60 , 60 , 60)

Now we can see that the vertex has been transformed correctly by the matrix into world space. This was made possible by first transforming (in this case translating) the vertex into bone space and then transforming it into world space. We can think of the transformation into bone space as attaching the vertex to the bone as a child. The vertex becomes defined as bone-relative, just as the frames in a hierarchy are defined relative to their own parents. Thus if the parent moves or rotates, so do the children. This is exactly the behavior we want from our vertex-bone relationship. Remember that in our X file, the skin mesh was not actually defined as a child in the local space of any frame (which was why it would have been fine to store it in any frame node – it was just for storage purposes). But in our last chapter of course, our non-skin scene mesh vertices were defined relative to the frames that they were in. So what we are doing with this transformation of the vertex to bone local space is an extra step to get us back to where we were in our last demo – where mesh vertices lived in frame local space.

Again, as we discussed in Chapter Four, to get object A into the local space of object B, assuming both objects currently exist in the same space (i.e. same frame of reference) we need to transform A by the inverse matrix of B. Note that in the case of our reference pose mesh, both the bone frame (once it has been concatenated) and the vertex are both defined in model space. Remember that the reference pose mesh has not been moved out into the world. That is, the root node frame matrix is identity.

If this all makes perfect sense to you, then feel free to skip the next section. However, if you still need a bit more insight into how this all works, please read on. We will go through this process a bit at a time and use some diagrams to try to make everything as clear as possible.

11.5.2 The Bone Offset Matrix: A Closer Examination

The mathematics of 3D transformations and all of the different 'spaces' can sometimes get a little confusing for new students. After all, just look at our last example of the vertex/bone transformation from the previous section:

Bone Space Vertex = v * -b = (60 , 60 , 60) + (-50 , -50 , -50)
= (10 , 10 , 10)

World Space Vertex = Bone Space Vertex * b = (10 , 10 , 10) + (50 , 50 , 50)
= (60 , 60 , 60)

It looks like all we have done is subtracted the bone's position from the vertex and then added it right back on again! But when you really think about it, it starts to make some sense. Since we were transforming the vertex using the skeleton in its reference pose, we expect that the resulting vertex we get back from the transformation is the vertex position we already have stored. After all, nothing has actually moved, so the relationships should be the same as they were before the transformation.

So let us go one step further. This time let us assume that our animation controller has assigned some transformation to bone **b** translating it from position (50,50,50) to (100,100,100) when all is said and done. We certainly want the vertex to correctly move when the bone that influences it moves. The system above does exactly that because we first subtract the bone's reference position from the model space vertex to transform the vertex into a bone local space position (i.e. the vertex is defined relative to the bone origin). We then use the newly modified combined bone matrix to transform the vertex into world space, thus maintaining the relationship between the vertex and the bone.

In the following formula, **b** is the bone's reference position (how it is stored in the X file's bone offset matrix) which never changes, **v** is the model space vertex, which also never changes and **c** is the current world transform (the combined matrix) of the bone that changes when animations are applied or when the character is moved about the world.

First we transform the vertex into bone space by subtracting the inverse ***reference*** position of the bone from the model space vertex.

Bone Space Vertex = v * -b (60 , 60 , 60) + (-50 , -50 , -50) = (10, 10, 10)

Now we transform the bone space vertex using the ***current*** world transform of the bone so that the vertex is correctly positioned in the world. In this example we are assuming the current world space position of the bone is (100,100,100) after animation and hierarchy matrix concatenation has occurred.

World Space Vertex = Bone Space Vertex * c = (10, 10 ,10) + (100, 100, 100)
World Space Vertex = (110 , 110 , 110)

As we can clearly see, the vertex has moved along with the bone. Although the bone has moved to position (100,100,100) the vertex has still maintained its relationship of a (10,10,10) offset from the bone, which placed it at world space position (110, 110, 110).

Now in these simple examples, we are assuming that the vertex is influenced by only one matrix. If the vertex is influenced by multiple matrices then the above steps would need to be performed for each matrix. The resulting temporary world space vertex positions would each be scaled by their respective weights before being added together to create the final world space vertex position, just as we saw earlier in our vertex blending code. It is also worthy of note that in the above example we are only showing the translation component of the process; the bone can also be rotated. So the fact that we have access to the bone offset matrix means that we always have the ability to transform the vertex back into bone space for a given bone and then apply the current world matrix of the bone. If the bone has been rotated, then the vertex will also be rotated about the bone space origin as expected rather than the model space origin.

So now we know that the X file will contain a bone offset matrix for every bone. We also know that when this matrix is applied to a model space vertex, it will transform the vertex from model space to bone space. This process essentially attaches the vertex to our bone as a child, just as we saw with our meshes in the last two chapters, which solves all of our problems. Remember that the bone offset matrices never change, while the world space bone positions do (when animated). With this in mind, we will rewrite our vertex blending function discussed earlier to first transform the vertex into each bone's space before applying the world transform for that bone.

> **Note:** You will not have to write your own vertex blending function when you are using the DirectX pipeline. You will just need to make sure that it has access to both the world matrix and the bone offset matrix for each bone for the proper vertex transformations.

Our function parameter list has now been modified from our earlier example to include a pointer to an array of bone offset matrices. We pass in each world space (combined) bone matrix in the pBoneMatrices array and we will also pass in each bone's corresponding bone offset matrix (loaded from the X file) in the pOffsetMatrices array. Notice that in this function, rather than applying the inverse transform matrix to the vertex and then applying the world bone matrix to the bone space vertex in two separate steps, we can combine the bone's world matrix with its offset matrix first to get a resulting matrix that will perform both steps with a single vertex/matrix multiply. So we will transform the vertex from model space straight into world space with this one combined matrix.

```
D3DXVECTOR3 TransformBlended ( D3DXVECTOR3* pModelVertex ,
                               D3DXMATRIX * pBoneMatrices ,
                               D3DXMATRIX *pOffsetMatrices ,
                               float *pWeights ,  float NumOfMatrices  )
{
      D3DXVECTOR3 TmpVertex;
      D3DXVECTOR3 FinalVertex ( 0.0f , 0.0f , 0.0f );
      D3DXMATRIX  mtxTempMatrix;

      for ( int I = 0 ; I < NumOfMatrices; I++ )
      {
            D3DXMatrixMultiply ( &mtxTempMatrix , pOffsetMatrices[I],
                                 pBoneMatrices[I]);
```

```
            TmpVertex = ( *pModelVertex * mtxTempMatrix ) * pWeights[I];
            FinalVertex += TmpVertex;
      }

      return FinalVertex;
}
```

Remember that the bone offset matrices never need to be recalculated. They are loaded (or created) during initialization and are re-used during each iteration of the game loop. When combined with the bone's world transform, we get a matrix that can correctly be used in blending operations to transform any vertices that are influenced by that bone.

As promised, before moving off the topic of the bone offset matrix, we will look at some additional examples with the aid of some diagrams to make sure that you fully understand the concept of the bone offset matrix.

In the following example we will see why the bone offset matrix is needed to transform a vertex that is influenced by two different bones in the hierarchy.

In Fig 11.14 we see two bones that influence a single vertex. For simplicity, we are working with 2D vectors. The absolute position of bone 1 in the skeleton reference pose is (5, 5) and the absolute position of bone 2 in the reference pose is (3, 8). The model space vertex in the reference pose is positioned at (5, 8). Although the bone matrices are initially stored as relative matrices in the X file, we are assuming in this example that the hierarchy has already been traversed and that the absolute position of each bone in the reference pose has been determined. We could say at this point that both of the bones and the vertex are now defined in the model space of the skin (assuming we have not moved the skeleton by updating the position of the root bone).

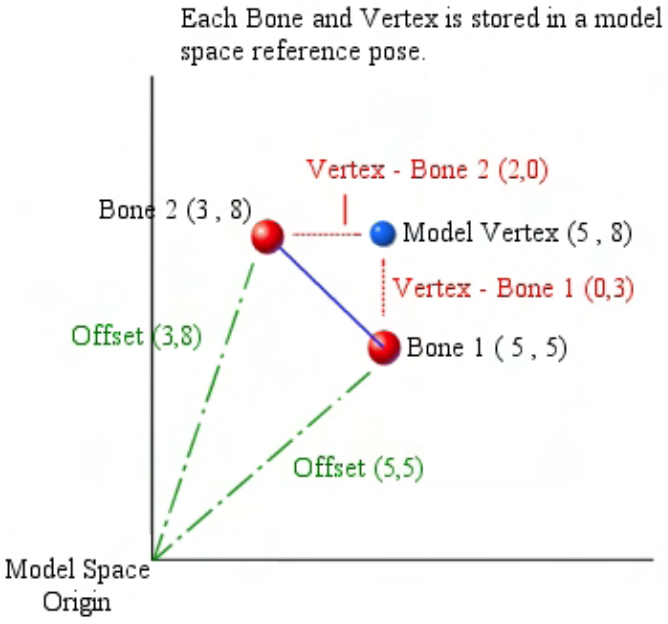


Figure 11.14

We note that the vertex is offset (0, 3) from bone 1 and (2, 0) from bone 2. We will assume that the weight of influence of each (bone) matrix on this vertex will be 0.5. That is, they will both be used in equal measure to transform the vertex into world space.

The two green lines in the diagram (labeled 'Offset') tell us how far from the model space origin each bone is offset. We can think of the bone offset matrix as being the reverse of these two lines. In other words, for bone 1, the offset matrix would have a translation vector of (-5, -5) and for bone 2 the bone offset matrix would have a translation vector of (-3, -8).

We know that in order to transform this model space vertex into world space we must transform it by each bone matrix, scale the resulting vectors by the respective weights, and then sum the results. If you recall our vertex blending function, you will remember that this is done one transformation at a time. We would first need to transform the vertex by bone 1 and then scale the result by bone 1's weight to create a temporary world space vertex for that matrix. We would then do the same for bone 2 and combine the results. So let us start our example with how the vertex would be transformed by bone 1.

For this example, we will discuss how to transform the vertex into world space *after* the two bones to which the vertex is attached are moved into a new world space positions.

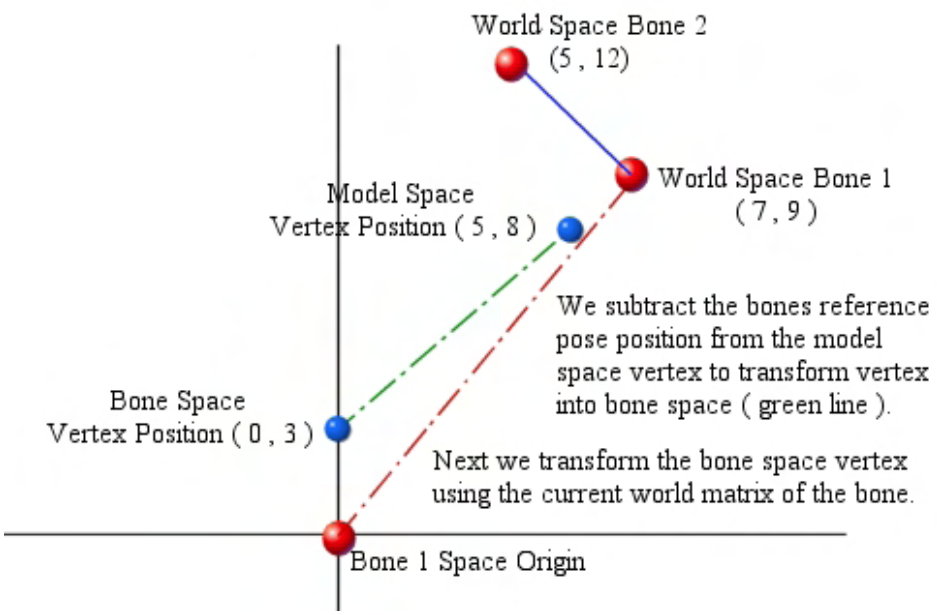


Figure 11.15

In Fig 10.15, both bone 1 and bone 2 are moved into world space positions (7, 9) and (5, 12) respectively. Beginning with bone 1 we apply the bone offset matrix to the model space vertex. This slides the blue vertex in the diagram from its model space position of (5, 8) into a bone 1 space position of (0, 3). As you can see, we have slid the vertex back along bone 1's offset vector. In bone 1 space we can imagine that bone 1 is now at the origin of the coordinate system and that the vertex is offset (0, 3) from the bone, as was the case in the reference pose.

Since the world transform for bone 1 is now (7, 9), we apply this transformation to the bone space vertex (0, 3) as shown in Fig 11.16.

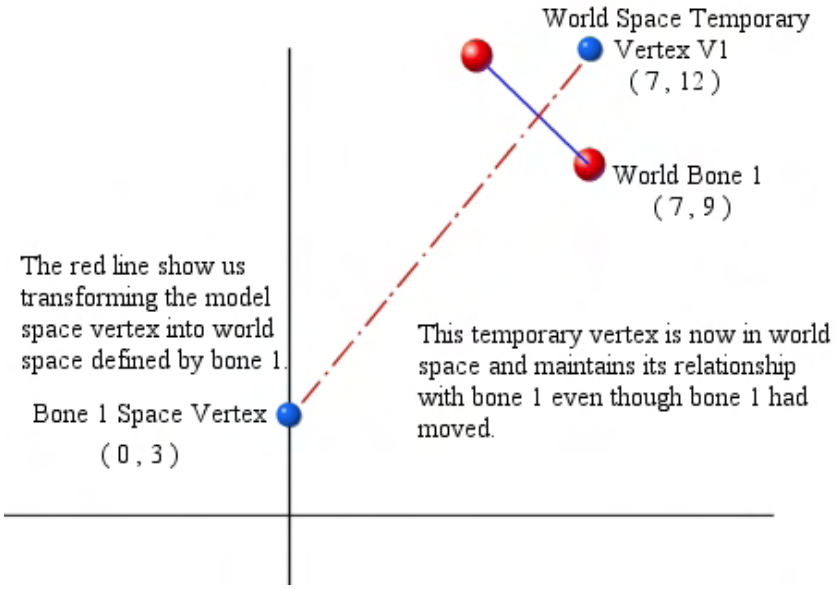


Figure 11.16

We can see in Fig 11.16 that we have now slid the vertex into position (7, 12) and it remains correctly offset (0, 3) from bone 1. So the relationship between the bone and the vertex in the reference pose has been correctly maintained even with the bone in its new world space position.

The next step is to scale the temporary world space vertex by the weight of bone 1 (0.5) for that vertex. This will effectively halve the length of the vector. In Fig 11.17 we can see that the final position of the temporary vertex, when multiplied by the weight of 0.5 (3.5, 6) is halfway along the red dashed diagonal line.

We now repeat the process all over again for bone 2. First we subtract the offset of bone 2 from the model space vector to get the vertex into bone space. We then multiply this vertex by the world matrix of bone 2 to create another temporary vertex of (7, 12) which is scaled by the weight of bone 2 (also 0.5) and we wind up with a temporary vertex position of (3.5, 6). Because both matrices influence the vertex equally, this means that we end up with two temporary vertex positions of (3.5, 6). When these two temporary vectors are added together we have the final world space position of the vertex (7, 12). Obviously, if the weights of the matrices for this vertex were altered so that one bone influenced the vertex more than the other and/or if the vertex was not positioned equidistant from each bone, the final world position of the vertex would be different.

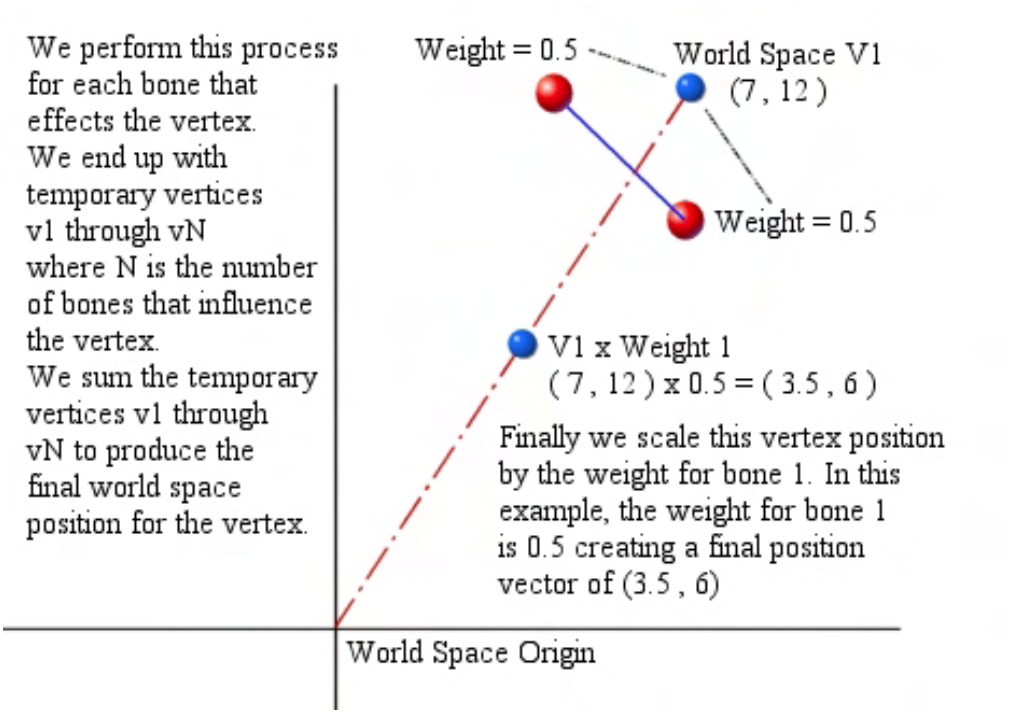


Figure 11.17

Bear in mind that we are showing a simple example here based on only translations. If any of the bones world matrices contained rotations, then the vertex would be rotated about the bone space origin before being translated into position.

Let us now tackle a slightly more complex example. In Fig 11.18 we see a model space vertex that has a position of (30, 20, 0) and is influenced by three bones. The diagram shows the position of each of the three bones in the reference pose and also shows the influence of each bone on the vertex. As you can see, the weights for this vertex are 0.25, 0.5, and 0.25 for bones 1 through 3, respectively. Therefore, we could say that bone 1 and bone 3 contribute 25% of the vertex world space position each, while bone 2 will contribute half with its influence at 50%. Once again, we are looking at the absolute bone positions for the reference pose calculated by traversing the hierarchy and combining the relative matrices stored therein.

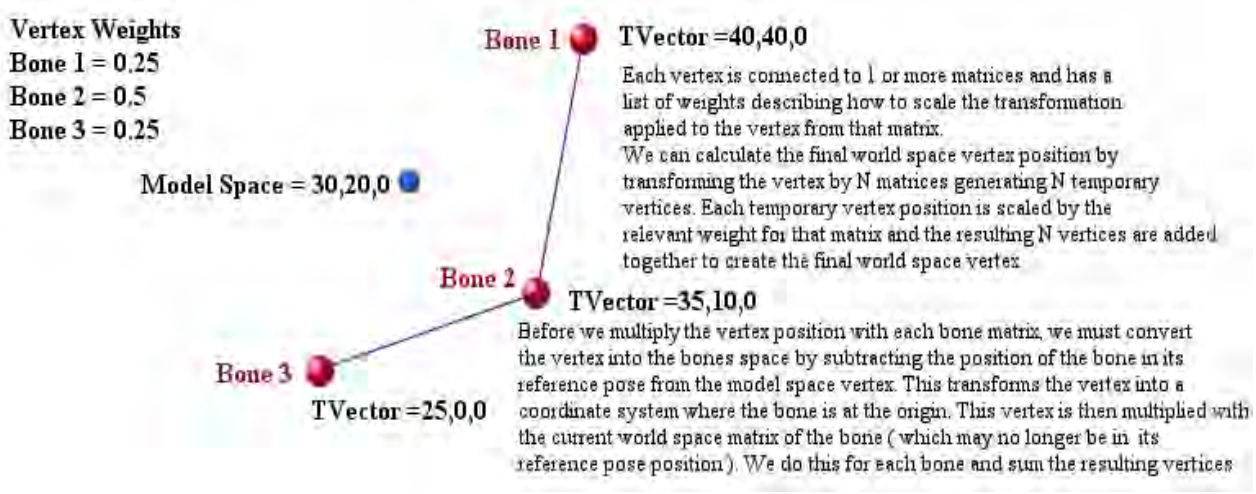


Fig 11.18

If we imagine that these three bones define the arm of a skeleton which is bent in its default pose, let us see what happens to the model space vertex when we move bone 3 to straighten the arm (Fig 11.19).

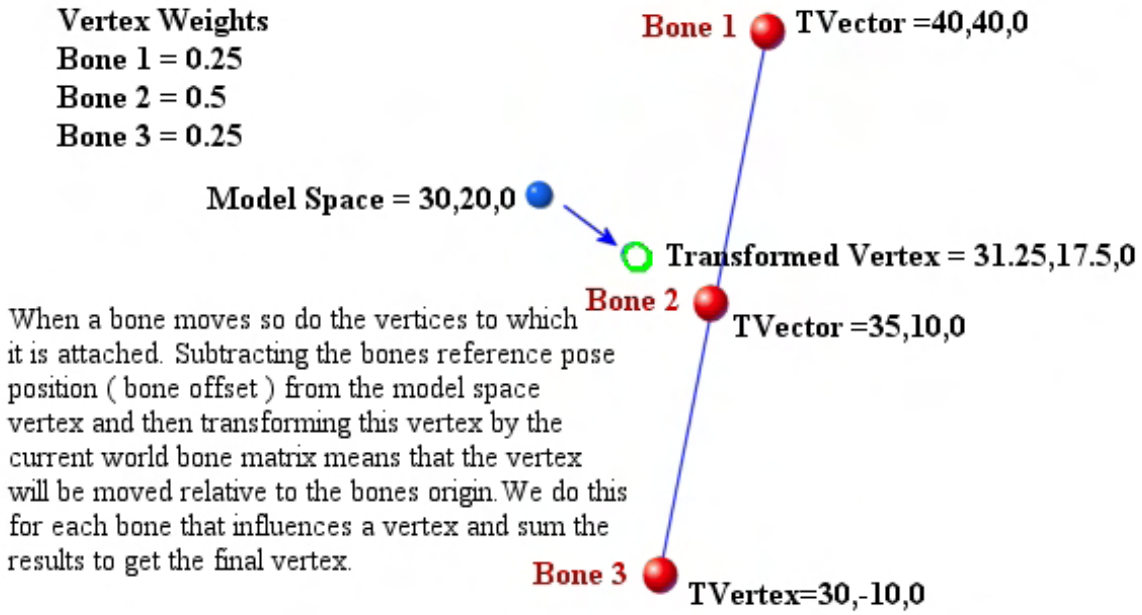


Fig 11.19

We already know that in order to calculate the final position of the vertex we must calculate (for each bone) a temporary vertex position by subtracting the bone's offset in the reference pose from the model space vertex. (Do not forget that technically this step is a bone offset matrix multiplication that accounts for rotation as well. We are focusing on translation only because the math is easier to grasp.) Then we have to transform the bone space vertex using the bone's combined world matrix and scale the result using the weight value. This will be done in three separate steps (once for each bone matrix) before summing the three temporary vertex positions to create the final world space vertex position.

The following calculations walk through the process. They demonstrate exactly what our vertex blending function would do if given the model space vertex, the three bone matrices, and their respective bone offset matrices from Figure 11.19.

Notice that for each of the three bones we transform the model space vertex into bone space using the current bone's inverse transformation (the bone offset matrix). We then transform the bone space vertex position with the current bone transformation and scale the resulting vector by the weight for that bone. Notice that in Fig 11.19, only bone 3 has been moved. Therefore, we expect that the transforms for bones 1 and 2 should just output the same model space vertex because the relationship between the vertex and those bones are the same as they were in the reference pose. However, when transforming the vertex by bone 3 (which has moved), the bone position is no longer equal to the reverse of the transformation described by the bone's offset matrix. Therefore, the vertex position in step three will not be the same model space vertex, and thus the sum of these vertices produces a different world space vertex position as well. The new world space vertex position is shown in Fig 11.19 as a green circle.

Transform Vertex V with Bone 1

Bone Space V ***= (30 , 20 , 0) + (- 40 , -40 , 0) = (-10 , -20 , 0)***
Un-Scaled World V1 ***= (-10 , -20 , 0) + (40 , 40 , 0) = (30 , 20 , 0)***
Scaled World V1 ***= (30 , 20 , 0) * 0.25 = (7.5 , 5 , 0)***

Transform Vertex V with Bone 2

Bone Space V ***= (30 , 20 , 0) + (-35 , -10 , 0) = (-5, 10, , 0)***
Un-Scaled World V2 ***= (-5 , 10 , 0) + (35 , 10 , 0) = (30, 20 , 0)***
Scaled World V2 ***= (30 , 20 , 0) * 0.5 = (15 , 10 , 0)***

Transform Vertex V with Bone 3

Bone Space V ***= (30 , 20 , 0) + (-25 , 0 , 0) = (5, 20, , 0)***
Un-Scaled World V3 ***= (5 , 20 , 0) + (30 , -10 , 0) = (35, 10 , 0)***
Scaled World V3 ***= (35 , 10 , 0) * 0.25 = (8.75 , 2.5 , 0)***

Sum the 3 temporary vectors to create the final vertex position

Final World Space Vertex = Scaled V1 + Scaled V2 + Scaled V3
= (7.5 , 5 , 0)
+ (15 , 10 , 0)
+ (8.75 , 2.5 , 0)

= (31.25 , 17.5 , 0)

Hopefully you now understand the purpose of the bone offset matrix. At its most basic level, it is used to temporarily attach the model space vertex to the bone hierarchy such that the vertex exists in bone local space before it is transformed into world space. This allows animation of the hierarchy to filter down to the level of each individual vertex in the skin because the vertex becomes a temporary child of the bone. Thus any movement of the bone will move the vertex, just as we saw in our last chapter when we noted that the transformation of a parent node affected the children of that node.

11.6 D3DXLoadMeshHierarchyFromX Revisited

We now know how all of the data for our skin and bone entities are stored in the X file. The next logical step then is to take a look at how we can load all of this information into our application. In this chapter we will use D3DXLoadMeshHierarchyFromX, but if for some reason you do not want to use this function, you should be in good position, given the information covered in this chapter and the last, to write code that parses the X file manually.

Because a skinned mesh is really just a mesh that has its vertices attached to various frames/bones in the hierarchy, we are not limited to storing only a single skinned mesh or skeleton per X file. It is entirely possible that the X file might contain a vast number of frames representing an entire scene, and embedded in that hierarchy might be several branches of the frame hierarchy that represent skeletal structures for several skinned meshes contained within that scene. After all, as far as the frame hierarchy and the animation controller are concerned, bones are just frames. The fact that a mesh may be contained within the file that uses some section of the frame hierarchy as a skeleton is insignificant in the grand scheme of things as far as the loading function is concerned. Of course, it is not insignificant to us since we would like to be able to separate the components in such a file according to the logical design requirements of our game engine. So when we load the various skinned meshes from the file, we will need to be able to identify the frames in the hierarchy that are being used as the mesh skeletons and make sure that we set everything up so that the appropriate meshes are associated with their skeletons and that their vertices are properly deformed during the frame animation step.

The D3DXLoadMeshHierarchyFromX function was covered in detail in the last chapter, but here is its definition again for convenience:

```
HRESULT D3DXLoadMeshHierarchyFromX
(
    LPCTSTR Filename,
    DWORD MeshOptions,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXALLOCATEHIERARCHY pAlloc,
    LPD3DXLOADUSERDATA pUserDataLoader,
    LPD3DXFRAME* ppFrameHeirarchy,
    LPD3DXANIMATIONCONTROLLER* ppAnimController
);
```

We know already how this function works; it returns the root frame in the hierarchy, which may be the root bone of a single skinned mesh skeleton or, if the file contains multiple skinned meshes or an entire scene, might just be the root frame of the scene itself and not mapped to any mesh vertices contained within. Either way, this is our handle to the entire scene and our means to traverse the frames/bones being used.

It is quite common (although not required) for each skinned mesh to be stored in a separate X file so that each call to D3DXLoadMeshHierarchyFromX creates a new hierarchy for each skinned mesh, with its associated animation data and its own animation controller. When this is the case, each frame hierarchy will contain the skeletal structure for a single mesh and its accompanying animation data. This is

generally the cleanest way to deal with skinned mesh characters and will be the method we prefer in this course.

Recall that if any animation data is stored in the X file, then an interface to a D3DXAnimationController object will also be returned. This will allow us to play back those animations to update the bones of our skeleton (and consequently, the skin). As discussed, the pAlloc parameter is only used when parsing custom data objects in the X file. Since we will not be requiring custom data in this chapter, we can assume that this value will be set to NULL.

So one of our first questions needs to be, how do we know when the loading function detects a skinned mesh in the file? A follow up question is, once identified, how do we setup and store that data for later use? The answers, as you may have already guessed, are to be found in the implementation of our ID3DXAllocateHierarchy derived class, and in particular our implementation of its CreateMeshContainer function. You should recall from the previous chapter that this function is called by the loading function whenever a mesh is encountered in the file. The mesh data is passed to this function allowing us to store it in the mesh container in the hierarchy as well as make any optimizations to the mesh data before we do so. It is in this function that we are informed that the mesh data we are being passed is intended to be used as a skin. We will be passed additional information about the mapping between the vertices of the mesh and the bones in the hierarchy in the form of an interface to an ID3DXSkinInfo object.

The ID3DXAllocateHierarchy::CreateMeshContainer definition is shown next for our convenience. It should already be quite familiar to you since we covered this function in detail in the previous chapter.

```
HRESULT CreateMeshContainer
(
    LPCSTR Name,
    LPD3DXMESHDATA pMeshData,
    LPD3DXMATERIAL pMaterials,
    LPD3DXEFFECTINSTANCE pEffectInstances,
    DWORD NumMaterials,
    DWORD *pAdjacency,
    LPD3DXSKININFO pSkinInfo,
    LPD3DXMESHCONTAINER *ppNewMeshContainer
);
```

This function is called for each mesh found in the X file. Our derived implementation is responsible for allocating a new D3DXMESHCONTAINER structure, taking the data passed into the function and storing it in this new mesh container, and then returning the newly populated mesh container back to the loading function. From there it will be attached to the relevant frame in the hierarchy. In Chapter Ten we saw that this was a relatively trivial task, but it was important because it allowed our application to own the memory for the meshes in the hierarchy. However, you might also remember that we ignored the LPD3DXSKININFO parameter – this will no longer be the case.

The pSkinInfo parameter is set to NULL when we are passed a normal (non-skin) mesh. For normal meshes, we simply store them in the mesh container and return as we did previously. If however, the pSkinInfo parameter is not NULL, then it means that the mesh we have been passed is to be used as a skin and it should have its vertices mapped to different bones in the hierarchy. When this is the case, we

have a fair bit of work to do. We have to make sure that we setup our mesh container so that we know which matrices have to be used to render a given subset in the mesh.

As it turns out, setting up a skinned mesh differs quite significantly depending on the type of skinning we are performing and whether or not hardware support is available. Since the different techniques can sometimes confuse matters when discussed collectively, what we will do in this chapter is talk about each skinning technique in isolation. On a case by case basis we will examine the skinned mesh creation process, the data structures needed to store it, and how to animate and render the skinned mesh.

In our lab project code we will implement all of the skinning techniques using a single class, but for now, we will consider each one separately for a more constructive analysis. This will make implementing any of the (three) methods we will study quick and easy. We will begin with the most basic skinning method -- software skinning. Essentially, this is a technique where the vertex blending stage is done in software (on the CPU) and not by the DirectX transformation pipeline on the GPU.

While examining each skinning technique we will look at how the CreateMeshContainer method should be written to correctly generate the skinned mesh for use with that technique.

11.7 DirectX Skinning I: Software Skinning

Software skinning is the first skinning technique that we will study in this chapter. As the name implies, it is not dependant on hardware support for vertex blending. As a result, it is not bound by the same limitations as hardware skinning, although it is generally not as fast. For example, when we use a hardware skinning technique, we are often limited to only a few bones being allowed to influence a single vertex. In the software case however we are not limited at all -- we can calculate our world space vertex positions using as many bone matrices as we desire. Therefore, while hardware skinning is preferable in most situations given the performance advantage, if you ever need to perform very complex skinning which requires large numbers of influential bones per vertex (more than would be supported by the DirectX transformation pipeline in hardware), software skinning would be your solution.

> **Note:** Even though software skinning is not as fast as hardware skinning, it still performs very respectably.

When we use software skinning, the skin vertex data will be stored in a standard ID3DXMesh. There is nothing special or unique about this mesh -- it will be passed to our CreateMeshContainer function and stored in our mesh container just as we saw in the previous chapter. During iteration of our game loop, the frame hierarchy is animated and we traverse it to build our absolute bone matrices. Our next task is to take the model space mesh vertex data we have stored in our mesh container and transform it into world space. We certainly do not want to overwrite the skinned mesh vertex data because this is the model space reference pose of the skin that we will need to transform against each time. Thus, an additional mesh will be needed to store the transformed results. We call this the *destination mesh*.

To render the skinned mesh, we will take the vertices in the original (*source*) mesh and transform them into world space (using multi-matrix blending) and store the resulting world space vertices in the vertex

buffer of the destination mesh. It is this destination mesh that we will render each frame. It is worth noting that because this destination mesh has its vertices already stored in world space, we should set the world matrix of the device to an identity matrix before we render the destination mesh.

So how do we transform the mesh from model space to world space when performing software skinning? Well, when a skinned mesh is found in the X file, our CreateMeshContainer function will be passed an ID3DXSkinInfo interface as well as the mesh for the skin. ID3DXSkinInfo contains lots of information, such as the bone offset matrix for each bone in the hierarchy and the mapping table which describes which vertices in the skin are mapped to which bones in the hierarchy and with what weight. We will store this ID3DXSkinInfo interface in our mesh container, so that we can access it in our game loop when we need to transform and render the mesh.

It just so happens that when performing software skinning, things could not be easier for us as programmers. Indeed, the ID3DXSkinInfo interface contains a helper function called UpdateSkinnedMesh which performs the entire transformation process for us. We will take a detailed look at this interface in a moment along with the many helper functions it provides, but first let us examine this vitally important method that is used to perform software skinning.

```
HRESULT UpdateSkinnedMesh
(
    CONST D3DXMATRIX *pBoneTransforms,
    CONST D3DXMATRIX *pBoneInvTransposeTransforms,
    PVOID pVerticesSrc,
    PVOID pVerticesDst
);
```

We pass this function the source mesh vertex buffer, our array of absolute bone matrices, and our array of bone offset matrices for each bone that influences the mesh. We also pass in the vertex buffer of a destination mesh where the transformed vertices will be stored. Because the ID3DXSkinInfo object contains all of the bone-to-vertex mapping information and the bone weight values, this function can quickly step through each vertex in the source mesh vertex buffer and use vertex blending (similar to the vertex blending code we introduced earlier) to transform each vertex using the relevant bones in the passed matrix arrays.

The resulting vertices will be copied to the destination vertex buffer (in a separate destination mesh that we will store in the mesh container structure). On function return, the destination mesh can then be used for drawing. Note that the source vertex buffer is read-only and is never modified by this function. We always maintain the original model space mesh. Every time the hierarchy is updated or animated, we must perform this step to calculate new values for the destination mesh. Again, because the destination vertex buffer that we render from always contains the mesh vertices in world space, we need to set the device world matrix to an identity matrix before rendering.

The way this function is designed does cause a small, but easily solved, problem for us. We need to pass in an array of absolute bone matrices as the first parameter, but our absolute bone matrices are not stored in array format – they are stored throughout the hierarchy in their respective frames. Therefore, in the CreateMeshContainer function, we will need to do two things. First, we will need to extract all of the bone offset matrices from the ID3DXSkinInfo and store them in an array inside the mesh container.

Second, we will need to fetch the corresponding absolute bone matrices in the hierarchy and store pointers in the mesh container as well. This means that our mesh container will store an array of pointers to absolute bones matrices and a corresponding array of bone offset matrices. Finding out which bone offset matrix belongs with which frame matrix in the hierarchy is the only tricky part, and we will examine how that is done momentarily, when we look at some code for an example CreateMeshContainer function.

> **Note:** While we could write our own vertex blending code, it is generally a better idea to use the ID3DXSkinInfo::UpdateSkinnedMesh function. On the whole, it performs vertex blending at a very respectable speed.

The basic steps for software skinning are shown next along with a detailed account of how we can implement this process.

1. Load and store the mesh as a standard mesh in the mesh container and store the passed ID3DXSkinInfo interface in the mesh container as well. This is all done inside the ID3DXAllocateHierarchy derived object's CreateMeshContainer method. We will also fetch the name of each bone stored in the ID3DXSkinInfo object and store pointers to those bones' absolute matrices in the mesh container. This gives us easy access to that information when transforming the mesh without having to traverse the hierarchy. We also retrieve the bone offset matrix for each bone from the ID3DXSkinInfo object and store them in the mesh container as well. In the end, we will have an array of bone matrix pointers and an array of their corresponding bone offset matrices ready to be sent to the UpdateSkinnedMesh method.

2. In our game loop, we animate the skeleton as usual using the animation controller (i.e. activate an animation set and call the AdvanceTime method). This will apply animation changes to the parent-relative matrices stored in each bone.

3. After calling AdvanceTime, we build the absolute (world) bone matrix for each frame by concatenating matrices as we step through the hierarchy. At the end of this stage, each bone in the hierarchy will contain an absolute world space matrix for that bone. It is these matrices that have pointers stored in the mesh container.

4. Now we call the ID3DXSkinInfo::UpdateSkinnedMesh function to transform our source mesh vertex data into a destination vertex buffer in a destination mesh. The destination vertex buffer will contain the updated world space positions of the skin's vertices. We pass this function the array of absolute bone matrices we have just updated and an array containing each bone's corresponding bone offset matrix. This function will combine each bone matrix with its corresponding bone offset matrix and use the resulting matrix to transform each vertex that is attached to that bone. The ID3DXSkinInfo object already stores the weight values, so it will manage the vertex blending calculations with little trouble.

5. We now have a destination mesh with the skin's world space vertices stored in its vertex buffer. We set the device world matrix to identity and then draw the mesh as usual -- loop through each subset, set the required textures and materials, and call DrawSubset. Remember that the destination mesh is just a regular ID3DXMesh.

Now that we know the basic steps involved in software skinning, let us look at some implementation details.

11.7.1 Storing the Skinned Mesh

Before we can render a skinned mesh we must be able to load it, and before we can do that we will need to setup some data structures to store the required information. This is what we will discuss in this section.

First, we will use the same D3DXFRAME derived structure that we used in the previous chapter. You will recall that we derived a structure from D3DXFRAME that contained an additional 4x4 matrix that to store the absolute world transform of each frame when we do our hierarchy update. We will now think of each frame as being a bone, and the frame hierarchy as the skeleton of our skinned mesh, but the core ideas remain the same.

```
struct D3DXFRAME_MATRIX: public D3DXFRAME
{
    D3DXMATRIX   mtxCombined;
};
```

The next structure that we will need to add new members to is the D3DXMESHCONTAINER. Let us first remind ourselves what the standard version of this structure looks like:

```
typedef struct _D3DXMESHCONTAINER
{
    LPTSTR              Name;
    D3DXMESHDATA        MeshData;
    LPD3DXMATERIAL      pMaterials;
    LPD3DXEFFECTINSTANCE pEffects;
    DWORD               NumMaterials;
    DWORD               *pAdjacency;
    LPD3DXSKININFO      pSkinInfo;
    struct _D3DXMESHCONTAINER *pNextMeshContainer;
} D3DXMESHCONTAINER, *LPD3DXMESHCONTAINER;
```

We will use this structure to store all of the necessary data for our skin. Unlike the previous chapter, we will now be using the pSkinInfo member to store the ID3DXSkinInfo interface that is passed to the CreateMeshContainer callback during the loading of the hierarchy. The ID3DXSkinInfo object will contain the names of all the bones used by this skin, the bone offset matrix for each of these bones, and the indices and weights describing which bones influence which vertices and to what extent. We do not have to manually create the ID3DXSkinInfo object -- it will be automatically created and populated by D3DXLoadMeshHierarchyFromX and passed to the CreateMeshContainer callback. All we have to do is store it and eventually use it to transform the mesh in our render loop. The rest of the members of this structure are used to store the ID3DXMesh (inside the MeshData member) and additional information such as materials and textures that are used by its various subsets.

Now we need to derive from this structure and add a few members of our own. For starters, we know that when we transform the model space source mesh into world space using ID3DXSkinInfo::UpdateSkinnedMesh, we will need a place to store the resulting vertices. Since we wish to maintain the original mesh data, we will add an ID3DXMesh to the mesh container called OriginalMesh. It will always store the mesh that was originally loaded and passed to the CreateMeshContainer function. Thus, we will use the mesh that is stored in the MeshData member of the mesh container as our destination mesh. MeshData will always contain the current world space transformation of the vertices and will be the mesh that we render each frame. Therefore, every time we call ID3DXSkinInfo::UpdateSkinnedMesh, the vertices from OriginalMesh will be transformed and stored in MeshData. This first new data member that we will add to our derived mesh container is shown below.

ID3DXMesh *pOriginalMesh

The next thing we have to consider is ease of access to the world matrices for each frame in the hierarchy. Recall that we need to pass this into UpdateSkinnedMesh as its first parameter. Theoretically, we could just copy the absolute bone matrices each time the hierarchy is traversed and updated, but that is cumbersome and time consuming. Instead, when we first create the mesh we will traverse the hierarchy and find all the bones used by the skin and store pointers to each of their absolute bone matrices in a bone matrix array. This array will be stored inside the mesh container so that when animation has been applied and the matrix for each bone has been updated, the mesh will have an array of pointers to these matrices. With these pointers on hand, we can build an array of bone matrices quickly that can be passed into the UpdateSkinnedMesh method. So the next new mesh container member is an array of matrix pointers:

D3DXMATRIX **pBoneMatrixPtrs

Note that this new member is an array of matrix pointers and not an array of matrices. This is very important, since we want each element in the array to point to the world matrix of each bone in the hierarchy used by this mesh. When the matrices are updated, these pointers will still point to them, so we will always have direct and efficient access to the absolute bone matrices used by this skin. They will not be invalidated when the bone matrices are recalculated.

We also know that before we use each absolute bone matrix to transform a vertex, it must be combined with the bone offset matrix first, so that transformations are performed in bone space. So in addition to our bone matrices, we must pass an array of corresponding bone offset matrices to the UpdateSkinnedMesh function. The function will automatically combine the matrices in these two arrays and use the resulting matrices to transform our skin vertices. Therefore, we will need to add another matrix array to our mesh container for the corresponding bone offset matrices. We will populate this array inside the CreateMeshContainer function (a one time process) with the bone offset matrices that we extract from the ID3DXSkinInfo object we are passed.

D3DXMATRIX *pBoneOffsetMatrices

Both the ppBoneMatrixPtrs array and the pBoneOffsetMatrices array should each have N elements, where N is the number of bones in the hierarchy used by the skin. We will see in a moment how the matrices will be stored such that pBoneOffsetMatrix[N] is the bone offset matrix for the bone stored in *pBoneMatrixPtrs[N]. This gives us a nice one-to-one mapping when it comes time to pass this data to the UpdateSkinnedMesh function.

We mentioned that the first parameter to UpdateSkinnedMesh should be an array of absolute matrices for each bone. The second parameter of course is an array of corresponding bone offset matrices. Perhaps you have already noticed the problem? Passing in the bone offset matrices is not a problem because we will already have these stored in the mesh container as an array (just as the function expects). But we do seem to have a problem with the absolute bone matrices. Right now, all we have is an array of matrix pointers, but the function expects an array of matrices.

So we will have to loop through all of the bones in the mesh container's bone matrix pointer array and copy the actual matrix data into a temporary array so that it can be passed to UpdateSkinnedMesh. Once we call UpdateSkinnedMesh, the original mesh will be transformed into the destination mesh and this combined matrix array will not be needed again until the next time we need to transform the mesh. Therefore, we will make this a global array that can be used as a temporary storage area for any skinned mesh that we are about to transform and render. Remember, these next two variables will not be part of the mesh container; they will be global variables. It is this temporary matrix array that we will pass to the UpdateSkinnedMesh function:

D3DXMATRIX *pgbl_CombinedBoneMatrices
DWORD gblMaxNumBones

Now this is where we will change the rules a bit. Since we have to loop through each element in our bone matrix pointer array and copy it into this temporary array before sending it into the UpdateSkinnedMesh function, we might as well combine each bone matrix with its corresponding bone offset matrix as we do so. Otherwise, UpdateSkinnedMesh will end up having to loop through each matrix array we pass and combine them anyway. So before calling UpdateSkinnedMesh, we will loop through each bone in our bone pointer array and combine each one with its corresponding bone offset array, storing the resulting matrices in this temporary array. As the matrices in this array are already combined, we can pass this array as the first parameter to UpdateSkinnedMesh and set the second parameter (the bone offset array parameter) to NULL. Thus, the matrix copy that we have to do is essentially free.

Our approach will be to allocate one global array that has enough elements to store the maximum number of bones used by any skinned mesh in our scene. When we are loading our skinned meshes from their X files, we will use a global variable called glbMaxNumBones to record the highest bone count so far so that we can make sure this global temporary array is large enough to accommodate any of our skinned meshes. Although this might sound complicated, it really is not -- this matrix array is just like a scratch pad where we combine each mesh's bone matrix with its corresponding bone offset matrix prior to passing it into the UpdateSkinnedMesh function. We will see it in use when we look at the code momentarily.

But first, let us have a look at our new derived mesh container:

```
struct D3DXMESHCONTAINER_DERIVED : public D3DXMESHCONTAINER
{
   ID3DXMESH   * OriginalMesh;
   D3DXMATRIX ** pBoneMatrixPtrs;
   D3DXMATRIX  * pBoneOffsetMatrices;
}
```

We now have proprietary data structures with everything we will need to store for software skinning. Populating these structures is done inside the CreateMeshContainer method of our ID3DXAllocateHierarchy derived object, called by D3DXLoadMeshHierarchyFromX for every mesh encountered in the X file. Let us next have a look at what this interface looks like, the information contained within the underlying object, and how we can use its member functions to access this information and store it in our mesh container structure. After that, we will take a look at one possible implementation of a CreateMeshContainer function to see how the mesh data is loaded and stored for software skinning.

11.7.2 The ID3DXSkinInfo Interface

For every mesh found in the X file that contains bone information, D3DXLoadMeshHierarchyFromX will create a new ID3DXSkinInfo object and pass its interface to our CreateMeshContainer callback. We will usually store this interface in the mesh container, especially in the case of software skinning where we will wish to use some of its member functions (such as UpdateSkinnedMesh). Besides having useful helper functions, it also contains vital information that we will need to retrieve. For starters, it will contain an array of all of the names of the bones in our skeleton to which vertices in the mesh are assigned. It also contains an array of bone offset matrices for each of those bone matrices. This is very important because the bone matrices themselves are stored in the hierarchy with their respective names, so this is our only means of identifying which bone offset matrix in the D3DXSkinInfo object belongs with (and should therefore be combined with) each bone matrix in the hierarchy. Therefore, one of the first things we will do with this interfaces is pair the bone offset matrices with their corresponding bones in the hierarchy. We will do this using the new matrix arrays that we added to our derived mesh container structure.

To accomplish this, we will first call the ID3DXSkinInfo::GetNumBones method to find out how many bones in the hierarchy are mapped to this mesh. Then we will implement a loop that iterates over this bone count and call ID3DXSkinInfo::GetBoneName to retrieve the name for each bone index. Once we have the name of the current bone, we will traverse our hierarchy and search for the frame with the same name. Once located, we will store a pointer to this frame's absolute matrix in our mesh container at ppBoneMatrixPtr[N] (N is the current loop counter). We will follow this with a call to the ID3DXSkinInfo::GetBoneOffsetMatrix function to return the bone offset matrix for bone N also. This matrix will be stored in pBoneOffsetMatrix[N] in our mesh container. At the end of the loop we will have a bone matrix pointer array and a bone offset matrix array in our mesh container that has a one-to-one mapping.

Before we transform the mesh we must loop through each bone in the mesh container's bone pointer array, and for the current iteration (N) we multiply *ppBoneMatrixPtr[N] by pBoneOffsetMatrix[N] to create a combined bone matrix that we store in our global array pCombinedBoneMatrix[N]. Once we have done this for each matrix, we can pass the pCombinedBoneMatrix array to ID3DXSkinInfo::UpdateSkinnedMesh function to transform our skin mesh into world space so that it is ready to be rendered.

This might sound complicated, but it can actually be done using only a few lines of code. This will be clear enough in a moment when we look at a simple example of a CreateMeshContainer implementation. But before we do that, let us first look at the member functions of the ID3DXSkinInfo interface that are applicable to software skinning. There are a few functions in this interface that are applicable only to hardware skinning, so we will postpone their coverage until later in the chapter. Since the D3DXSkinInfo object is really just an object that encapsulates all of the SkinWeights data objects found in the X file for a given mesh, most of the functions are used to simply retrieve or modify this vertex/weight/bone data.

ID3DXSkinInfo Member Functions

`HRESULT Clone( LPD3DXSKININFO *ppSkinInfo );`

Given a pre-populated ID3DXSkinInfo interface, we can use this function to clone the object. We pass in the address of an ID3DXSkinInfo interface pointer and on function return it will point to the newly created object. All of the bone name, bone offset matrix, and vertex weight data contained in the original skin info object will be copied into the clone.

`DWORD GetNumBones(VOID);`

This function returns the number of bones in the hierarchy referenced by this skin. The bone matrices themselves are not stored in this object (they are stored in the frame hierarchy) but this object does contain a list of bone names and their respective bone offset matrices. We can use the bone names to pair the bone offset matrices with their matching bone matrices in the hierarchy. In our lab projects, the value returned by this function will be used to allocate the matrix arrays we have added to our D3DXMeshContainer structure.

`LPCSTR GetBoneName( DWORD Bone );`

This function returns the name of the bone stored at the index specified by the input parameter. We will use this function when setting up our mesh container inside the CreateMeshContainer function to get the name of each bone and match it to a bone in the hierarchy. We can then store this bone matrix and its accompanying bone offset matrix in the mesh container.

`LPD3DXMATRIX GetBoneOffsetMatrix( DWORD Bone );`

This is the function we will use to retrieve the bone offset matrix for each bone in the hierarchy. Our code will first use the GetNumBones member function to find out how many bones influence this skin. This count value also defines the number of bone offset matrices (and bone names) stored in the skin info object's internal arrays. We can loop through each of these bone indices and call GetBoneName to fetch the current bone name and GetBoneOffsetMatrix to fetch the 4x4 bone offset matrix that belongs with that bone name. Once we have the bone name and bone offset matrix, we can traverse the hierarchy

to find the matching frame in the hierarchy to access its bone matrix. After we have both a pointer to the bone matrix in the hierarchy and the bone offset matrix, we can then store them in the mesh container in their corresponding data arrays (ppBoneMatrixPtrs and pBoneOffsetMatrices).

> **Note:** Writing code to find a bone in the hierarchy would be simple enough to do. But it is even easier if we use the D3DXFrameFind function. We will see this in a moment.

```
HRESULT GetBoneInfluence(DWORD Bone, DWORD *vertices, FLOAT *weights);
```

This function allows us to retrieve the indices of the vertices in the mesh that are attached to the bone specified in the first parameter, along with the weights for each of those vertices. When using software skinning, we will not have to worry about the weight data for each vertex/bone combination because this information is stored inside the ID3DXSkinInfo object. The ID3DXSkinInfo::UpdateSkinnedMesh function will automatically use the stored vertex weights to perform the vertex blending. This function would be useful if you are performing hardware vertex blending and you need to extract the weight information for each vertex and store it in the vertex structure (but there is even a helper function that does that for us as we will see later in the chapter). This function might also be useful diagnostically if you need to output or log this information.

The first parameter is the index of the bone that we would like the vertex/weight data returned for. The second and third parameters are pointers to pre-allocated arrays that should be large enough to hold the vertex index and vertex weight data. Each element in the DWORD array describes the index of a vertex in the skin that this bone is attached to. There is a corresponding weight defined in the float array describing how much that vertex is influenced by the bone matrix. The amount of memory that we must allocate for these arrays can be determined by calling the next function we will examine (GetNumBoneInfluences).

```
DWORD GetNumBoneInfluences( DWORD bone );
```

This function is used to find out how many vertices are attached to a bone. We pass in the index of the bone that we would like to enquire about and we are returned the attached vertex count. So if the bone in question influenced four vertices, the return value would be 4. We can use this value to allocate the vertex index and weight arrays passed in as the second and third parameters to GetBoneInfluence. We will not need to use this function during software skinning but it might be useful for diagnostic purposes.

```
HRESULT GetMaxVertexInfluences( DWORD *maxVertexInfluences );
```

This function can be used if you wish to enquire about the maximum number of vertices attached to a given bone in the hierarchy. We pass in the address of a DWORD which, on function return, will be filled with this value. For example, if this value was 3 on function return, then it means that at least one of our bones (but perhaps many) has three vertices attached to it and that there are no other bones that have more than this number of vertices attached to them.

```
FLOAT GetMinBoneInfluence(VOID);
HRESULT SetMinBoneInfluence( FLOAT minInfluence );
```

These functions allow us to get and set the minimum weight thresholds used to determine how much a bone matrix will contribute towards the transformation of one of its attached vertices. For example, if we set this value to 0.2 and then called UpdateSkinnedMesh, any matrices that influence a vertex with a weight less than 0.2 will not be used in the vertex blend. This might be useful if you know that you have some bones mapped to vertices with very small weights which do not influence the vertex position to any noticeable degree. You could set this threshold value so that these matrices will be ignored when transforming that vertex, speeding up the software transformation process. You will only rarely need to use this functionality (we will not use it at all in our lab projects).

```
HRESULT GetMaxFaceInfluences(LPDIRECT3DINDEXBUFFER9 pIB,
                             DWORD NumFaces, DWORD *maxFaceInfluences );
```

This function returns the maximum number of bones that affect a single triangle in the mesh. We pass in the index buffer of the mesh we are testing as the first parameter and the number of triangles in the buffer as the second parameter. The third parameter is the address of a DWORD which on function return will store the value of the maximum number of matrices that affect a single triangle in the mesh.

When performing software skinning, this information is of little significance since we are not limited to only using a few matrices per face. But when we discuss hardware skinning later in the chapter we will see that things are very different. For example, a given graphics card might only allow the setting of 6 bone matrices on the device at any given time, even though it is possible that each vertex in a triangle may be influenced by as many as 4 bones. This means that a single triangle could be influenced by 12 unique matrices and to be correctly transformed and rendered by the 3D pipeline we would need the ability to set those 12 matrices. This cannot be done on the hardware in our example so we would have to place the device in software vertex processing mode where the limitation is removed (we must have a software or mixed mode device to be able to do this). Note that the pipeline will still perform the vertex blending for us, but it will be done by the DirectX software transformation module. So we would lose the advantages of hardware T&L for that mesh. Please note that this is not the same as software skinning (which we are currently discussing) where the pipeline will not have the vertex blending render state enabled at all. In other words, when performing software skinning, the pipeline does not multi-matrix transform (i.e., vertex blend) our vertices. We are instead using a function (UpdateSkinnedMesh) to do our vertex blending outside of the pipeline. We will discuss pipeline vertex blending capabilities and hardware skinning later in the lesson.

```
HRESULT UpdateSkinnedMesh
(
    CONST D3DXMATRIX *pBoneTransforms,
    CONST D3DXMATRIX *pBoneInvTransposeTransforms,
    PVOID pVerticesSrc,
    PVOID pVerticesDst
);
```

This is the function that performs vertex blending when using software skinning. It transforms the model space vertices in the source mesh into vertex blended world space vertices in the destination mesh.

This function is called during the render loop to update the skinned mesh vertices whenever any of its bones have been animated or changed. The first parameter is an array of bone matrices (the absolute/world space frame matrices) and the second parameter is an array containing the corresponding bone offset matrices. The third parameter is the vertex buffer from the source mesh which contains the model space vertices of the skin in its reference pose (as passed into the CreateMeshContainer function by D3DXLoadMeshHierarchyFromX). The fourth parameter is the vertex buffer of the destination mesh that will receive the world space vertices and will be used to render the mesh in its current position and orientation. This function first combines each bone matrix in the pBoneTransforms array with the corresponding bone offset matrix in the pBoneInvTransposeTransforms array to generate the final world space matrix for each bone. It then multiplies each vertex with the bone matrices that influence that vertex to generate the blended vertex for the final vertex buffer. If the vertex format contains a normal, then the normals will also be blended in the same way, but without the translation portion of the matrix being used.

Recall that in our code we will allocate a third (global) matrix array called pCombinedBoneMatrices to store the combined bone matrices prior to passing them to this function. We do this because our mesh container stores an array of pointers to the bone matrices and not an array of the bone matrices themselves. In order to pass the bone matrices into this function, we would first need to copy them into a temporary array. Since we are looping through each bone anyway for copying purposes, we might as well just do the multiplication ourselves and store the result in the pCombinedBoneMatrices array. Since this global array will now hold the combined bone matrices, we can simply pass this array in as the first parameter to UpdateSkinnedMesh and set the second parameter to NULL. The function will still work perfectly; it will simply skip the matrix concatenation step that we just performed (thus making our matrix copy a cost-free operation).

11.7.3 Manual Skin Creation

It is possible to create an empty ID3DXSkinInfo and populate it programmatically or with skinning information stored in a custom file format. For example, you might have a mesh object that was not originally intended as a skin (or perhaps it was, but it was not provided in skin format). You can programmatically build your own frame hierarchy to represent the bones of the skeleton and then write some code that maps that skeleton to the mesh. Alternatively you might start with a skeleton and then procedurally build a skin to fit around it. Once done, you could create an empty ID3DXSkinInfo and populate it with the information for each bone, such as its name in the hierarchy, which vertices are influenced by that bone, and the bone weights. You would also need to calculate the bone offset matrix for each bone (which we learned how to do earlier) and store those as well.

The ID3DXSkinInfo interface has three 'Set..' functions which can be used to populate an empty D3DXSkinInfo object. Before we look at these, let us first talk about the global D3DX function that can be used to create an empty D3DXSkinInfo object. Although you will not need to use these functions when you are loading your data from X files, they are extremely useful if you are building your skinned meshes by hand. While we will not spend time in this chapter doing this, it is an important technique to learn. So in the next chapter, we will spend a good deal of time examining how to construct a skinned mesh manually to create some interesting animating trees for our outdoor environments. For now, we will just get an overview of the functionality involved.

```
HRESULT D3DXCreateSkinInfoFVF(DWORD numVertices, DWORD FVF,
                              DWORD numBones,
                              LPD3DXSKININFO* ppSkinInfo );
```

Although ID3DXSkinInfo does not store the actual vertex data (this is stored in a separate ID3DXMesh), the first parameter should contain a value describing the number of vertices in the mesh that we intend to use as the skin and the second parameter should be the FVF flags for the vertex structure we intend to use. This is important, because the UpdateSkinnedMesh function will need to know if the vertex contains a normal so that the normal is also blended when the vertex is transformed. The third parameter is the number of bones that this skin will need to be mapped to (i.e. the number of bones in the hierarchy for which we intend to attach a vertex). The D3DXCreateSkinInfoFVF function will use this number to allocate an internal bone name array and a bone offset matrix array of the correct size. The fourth parameter is the address of an ID3DXSkinInfo interface pointer which on successful function return will point to a valid interface to an empty D3DXSkinInfo object.

Once we have created an empty ID3DXSkinInfo object we need to populate it with data -- filling in the name of each bone, the vertex indices and weights for that bone and the bone offset matrix for that bone.

```
HRESULT SetBoneName( DWORD BoneNumber, LPCSTR BoneName );
```

This function is used to set the name of a bone. The first parameter is the index of the bone we wish to set the name for. For a newly created object, this will start off at 0 and increase as we add each bone's information. This index should never be greater than or equal to the numBones parameter passed into

the D3DXCreateSkinInfoFVF function. The BoneName we pass in should also match the name of one of the bones in the hierarchy.

```
HRESULT SetBoneInfluence(DWORD Bone, DWORD numInfluences,
                         CONST DWORD *vertices, CONST FLOAT *weights);
```

Once you have set the bone name, you have to tell ID3DXSkinInfo which vertices in the skin mesh are attached to this bone and the weight at which this bone should be used to influence each of them. The first parameter is the index of the bone we are setting the vertex mapping for. This should be followed by a DWORD parameter describing how many vertices in total will be influenced by this bone. As the third parameter we pass in an array of DWORDs where each element in the array describes the index of a vertex in the skin that will be influenced by this bone to some degree. The fourth parameter is where we store the matching weights for each vertex index specified in the previous parameter.

```
RESULT SetBoneOffsetMatrix( DWORD Bone,LPD3DXMATRIX pBoneTransform );
```

This function allows us to set the bone offset matrix for each bone. This matrix can be calculated by traversing the reference pose hierarchy (concatenating as we go) to the bone in question and then taking the inverse of the concatenated matrix.

At this point we have a through enough understanding of what the ID3DXSkinInfo interface contains and how to extract that data to store it in the mesh container to move forward. We also know how to manually create and populate an ID3DXSkinInfo if we need to fill it with some other non-X file data. Finally, it is worth remembering that, in the software skinning case at least, once we have extracted this information we do not just want to release the ID3DXSkinInfo interface. Instead we want to store it in the mesh container because we will use its UpdateSkinnedMesh function to transform the vertices of the skin into world space.

Let us now see how all of this theory can be put into practice by looking at a simple CreateMeshContainer implementation.

11.7.4 The CreateMeshContainer Function

In Chapter Nine we learned how to derive a class from the ID3DXAllocateHierarchy interface and implement its CreateMeshContainer function. This function will be called for each mesh contained in the X file. If the mesh is a skinned mesh, then the function will be passed an ID3DXMesh containing the skin (this is just a regular mesh) and an ID3DXSkinInfo interface that will contain the bone to vertex mapping information. Our function will need to store the mesh and the ID3DXSkinInfo interface pointer in the mesh container and will need to extract the bone offset matrices and store them as well. Our other responsibility is to find the matching bone matrices in the hierarchy and store pointers to them in our mesh container. Let us now have a look at the code to a typical function that would do all of this.

In this next example we will not show the processing of the material or texture data since we already know how to do this. The more complete function is of course fully implemented in the source code that accompanies this chapter. It is important to note that this implementation of the CreateMeshContainer function is for the software skinning case only. We will look at how this function will need to be implemented for the hardware skinning cases later (in our lab projects we will write a single function that handles all cases).

The function starts by allocating a new mesh container structure and getting a pointer to the ID3DXMesh passed into the function. This mesh will contain the skin in its model space reference pose.

```
HRESULT CAllocateHierarchy::CreateMeshContainer
       ( LPCTSTR Name, LPD3DXMESHDATA pMeshData, LPD3DXMATERIAL pMaterials,
         LPD3DXEFFECTINSTANCE pEffectInstances, DWORD NumMaterials,
         DWORD *pAdjacency,
         LPD3DXSKININFO pSkinInfo, LPD3DXMESHCONTAINER *ppNewMeshContainer )
{
    D3DXMESHCONTAINER_DERIVED *pMeshContainer = NULL;
    UINT iBone, cBones;
    LPDIRECT3DDEVICE9 pd3dDevice = NULL;
    LPD3DXMESH pDestinationMesh = NULL;
    *ppNewMeshContainer = NULL;

    // Get a pointer to the mesh of the skin we have been passed
    pDestinationMesh = pMeshData->pMesh;

    // Allocate a mesh container to hold the passed data.
    // This will be returned from the function
    // where it will be attached to the hierarchy by D3DX.
    pMeshContainer = new D3DXMESHCONTAINER_DERIVED;
    memset(pMeshContainer, 0, sizeof( D3DXMESHCONTAINER_DERIVED ));
```

Next we copy over the name of the mesh passed into the function and store it in the mesh container. We also get a pointer to the device object to which the mesh belongs as we will need it in a moment.

```
    // If this mesh has a name then copy that into the mesh container too.
    if ( Name ) pMeshContainer->Name = _tcsdup( Name );

    pDestinationMesh->GetDevice(&pd3dDevice);
```

The next section of code is optional; it is not specific to skinned meshes and applies to meshes of any kind. It is shown here for completeness. It tests to see if the input mesh has vertex normals defined. If not, then in this example we will assume that they are desired and clone the mesh to create a new skin that has room for vertex normals. We store the cloned result in the mesh container's MeshData member. After cloning, we create normals using the very handy D3DXComputeNormals helper function.

```
    // if no normals exist in the mesh, add them ( we might need to use lighting )
    if (!(pDestinationMesh->GetFVF() & D3DFVF_NORMAL))
    {
        pMeshContainer->MeshData.Type = D3DXMESHTYPE_MESH;

        // Clone the mesh to make room for the normals
        pDestinationMesh->CloneMeshFVF( pMesh->GetOptions(),
```

```
                                        pMesh->GetFVF() | D3DFVF_NORMAL,
                                        pd3dDevice,
                                        &pMeshContainer->MeshData.pMesh );

         pDestinationMesh = pMeshContainer->MeshData.pMesh;

        // Now generate the normals for the pmesh
        D3DXComputeNormals( pDestinationMesh, NULL );
    }
```

In the above code, we clone the mesh and divert the pDestinationMesh pointer to point at the newly created mesh interface. Remember, this originally pointed at the mesh passed into the function by D3DX which will no longer be used (because it did not have normals). We did not increment the reference count of the mesh when we assigned this pointer and thus do not need to release it here (it was a temporary pointer assignment). When this function returns, D3DX will release its hold on the original mesh interface causing it to be unloaded from memory. This is exactly what we want since we will prefer to use the newly cloned mesh instead.

If the mesh already contained normals, then we will use this mesh as is and simply store the passed mesh in the mesh container's MeshData member. We are careful to increment the mesh's reference count so that it is not unloaded from memory on function return when D3DX releases its claim on the object.

```
    else  // if normals already exist, just add a reference
    {
        pMeshContainer->MeshData.pMesh = pDesinationMesh;
        pMeshContainer->MeshData.Type = D3DXMESHTYPE_MESH;
        pDestinationMesh->AddRef();
    }
```

At this point in the function the mesh has been copied into the mesh container and contains vertex normals -- so far we have done nothing that we would not do for a standard mesh.

The next section of code is where we identify whether or not this is a skinned mesh. If the pSkinInfo parameter is not NULL, then this is indeed a skinned mesh and we will need to fill out the new members of our mesh container structure. Before we do this, we would probably store the textures and materials as shown in previous lessons, but we will skip that step for this example to keep things simple.

If this is a skinned mesh then we will copy the passed ID3DXSkinInfo interface pointer into the mesh container and make sure that we increment the reference count (we do not want the object prematurely destroyed when the function returns).

```
    // If there is skinning information, save off the required data
    // and then set up for skinning because this is a skinned mesh.
    if (pSkinInfo != NULL)
    {
        // first save off the SkinInfo and original mesh data
        pMeshContainer->pSkinInfo = pSkinInfo;
        pSkinInfo->AddRef();
```

The next step is to find out how many bones this skin is influenced by, using the ID3DXSkinInfo interface GetNumBones function. Once we have this value we know how large to make the mesh container's bone offset matrix array. After we allocate the bone offset matrix array, we loop through each bone and extract its bone offset matrix using the ID3DXSkinInfo::GetBoneOffsetMatrix member function and store the matrix in the mesh container's array.

```
        // We now know how many bones this mesh has
        // so we will allocate the mesh containers
        // bone offset matrix array and populate it.
        NumBones = pSkinInfo->GetNumBones();
        pMeshContainer->pBoneOffsetMatrices = new D3DXMATRIX  [NumBones];

        // Get each of the bone offset matrices so that
        // we don't need to get them later
        for (iBone = 0; iBone < NumBones; iBone++)
        {
             pMeshContainer->pBoneOffsetMatrices[iBone] =
           *(pMeshContainer->pSkinInfo->GetBoneOffsetMatrix(iBone));
        }
```

You might assume that we should now set up the mesh container's ppBoneMatrixPtrs array so that we can store all of the pointers to the absolute bone matrices in the hierarchy. However, at this point the hierarchy has not yet been fully created, so this is something that we will have to do after we have loaded the entire hierarchy and the D3DXLoadMeshHierarchyFromX function has returned program flow back to our application.

What we can do now though is check how large our temporary global combined matrix array is going to be because it needs to be able to hold enough matrices to accommodate the skinned mesh that uses the largest number of bones. So we check the current size of this global array (which will initially be allocated to a small default size) and if it is too small to contain the number of bones used by the mesh we are loading, we have found a new largest bone count and we resize the array. After all skinned meshes have been loaded, this array will be large enough to hold the bones of any skinned mesh in the scene. Recall that this global array is used by our application as a scratch pad prior to transforming a skinned mesh. It will be used to hold an array of concatenated bone and bone offset matrices for the mesh which will be passed into the ID3DXSkinInfo::UpdateSkinned Mesh function and used to multi-matrix transform the vertices.

```
       // Resize the global scratch array if this mesh
       // has more bones then any others previously loaded
       if ( gblMaxNumBones < NumBones)
        {
            gblMaxNumBones = NumBones;

            // Allocate space for blend matrices
            delete [] pgbl_CombinedBoneMatrices;
            pgbl_CombinedBoneMatrices  = new D3DXMATRIX [ NumBoneMatricesMax ];
        }
```

At this point our mesh container stores the model space reference pose mesh, the ID3DXSkinInfo interface, and all of the bone offset matrices. As discussed earlier, our preference will be to use the

MeshData member of the mesh container to store our destination mesh (i.e., the render-ready world space mesh), but right now it currently contains the original model space vertex data (i.e., the source mesh). So we will clone the mesh and store a pointer to the clone in the pOriginalMesh member that we added to our mesh container. Unlike its MeshData sibling, this mesh will never be overwritten and will always contain the model space mesh reference pose. This is the mesh whose vertex buffer will be passed into UpdateSkinnedMesh as the source vertex buffer. The transformed vertices will be copied into the destination mesh stored in the MeshData member of the mesh container for later rendering.

```
       // Finally clone the mesh so that we now
       // have a copy of it stored in the 'OriginalMesh' member
       // of the mesh container.
      // This is the mesh that will be the source mesh in transformation
       pDestinationMesh->CloneMeshFVF( D3DXMESH_MANAGED,
                                       pMeshContainer->MeshData.pMesh->GetFVF(),
                                       m_pd3dDevice,
                                      &pMeshContainer->pOriginalMesh );
  }
```

As it turns out, these are the only steps we will need to take in our CreateMeshContainer function when we are creating a mesh for software skinning.

Finally, we assign the passed mesh container pointer to point at our newly created mesh container so that when the function returns, D3DXLoadMeshHierarchyFromX will have access to it and will be able to attach it to the hierarchy.

```
    // Return the transformation
    *ppNewMeshContainer = (D3DXMESHCONTAINER *)pMeshContainer;
    pMeshContainer = NULL;

    pd3dDevice->release();
    return D3D_OK;
}
```

11.7.5 Fetching the Bone Matrix Pointers

After we have loaded our scene using D3DXLoadMeshHierarchyFromX, any skinned meshes that were contained inside the X file are now in their respective mesh containers in the hierarchy. However, our mesh container also has a new member called ppBoneMatrixPtrs which stores pointers to each absolute bone matrix for easy access after hierarchy changes. After we have finished loading the hierarchy, our next task will be to allocate this array so that we can populate it with the relevant matrix pointers. Our scene initialization code will usually look something like the code shown next. Note that this code assumes that CAllocateHierarchy is the name of our ID3DXAllocateHierarchy derived class and that pDevice is a pointer to a valid Direct3D device object.

```
D3DXFRAME  pFrameRoot;
ID3DXAnimationController *pAnimController;
CAllocateHierarchy Alloc;
D3DXLoadMeshHierarchyFromX("MyFile.x", D3DXMESH_MANAGED,
```

```
                              pDevice, &Alloc, NULL, &pFrameRoot,
                             &m_pAnimController);

SetupBoneMatrixPointers ( pFrameRoot , pFrameRoot );
```

The SetupBoneMatrixPointers function will be a simple recursive function that we write to finalize our mesh container data. In our actual code for this lesson we will store each loaded hierarchy inside a separate CActor class, for ease of data management. So if we loaded three X files, each one containing a separate skinned mesh, we would have three CActor objects in our application -- each representing a different character with its own root frame and its own animation controller. We will also make the SetupBoneMatrixPointers function we are about to discuss a member function, but for now we will forget about this encapsulation and just look at the techniques involved. We will call the following function for each skeletal hierarchy that we load:

```
HRESULT SetupBoneMatrixPointers(LPD3DXFRAME pFrame , LPD3DXFRAME pRoot)
{
    if (pFrame->pMeshContainer != NULL)
         SetupBoneMatrixPointersOnMesh ( pFrame->pMeshContainer, pRoot );

    if (pFrame->pFrameSibling != NULL)
         SetupBoneMatrixPointers ( pFrame->pFrameSibling , pRoot  ) ;

    if (pFrame->pFrameFirstChild != NULL)
         SetupBoneMatrixPointers ( pFrame->pFrameFirstChild , pRoot );

    return S_OK;
}
```

This function recursively traverses each frame (bone) in the hierarchy and tests to see if the frame has a mesh container attached to it. If it does, then it calls the SetupBoneMatrixPointersOnMesh function to set up the matrix pointers in the mesh container. We will look at that function in a moment. After SetupBoneMatrixPointers has returned control back to the caller, all of the mesh containers which store skins will have had their ppBoneMatrixPtrs array allocated and populated with pointers to each absolute bone matrix in the hierarchy. Notice that SetupBoneMatrixPointers passes the root frame pointer down the tree as well so that it is accessible in the SetupBoneMatrixPointersOnMesh function. This allows us to search the hierarchy starting at the root for a specific bone.

Let us now examine the SetupBoneMatrixPointersOnMesh function.

```
HRESULT SetupBoneMatrixPointersOnMesh( LPD3DXMESHCONTAINER pMeshContainerBase ,
                                       D3DXFRAME *pRootFrame  )
{
    UINT iBone, NumBones;
    D3DXBone *pFrame;

    D3DXMESHCONTAINERDERIVED *pMeshContainer =
                                    (D3DXMESHCONTAINER_DERIVED*)pMeshContainerBase;


    // If this is a skinned mesh then setup its bone matrix pointers
```

```
    if (pMeshContainer->pSkinInfo != NULL)
    {
        NumBones = pMeshContainer->pSkinInfo->GetNumBones();
        pMeshContainer->ppBoneMatrixPtrs = new D3DXMATRIX*[cBones];

        for (iBone = 0; iBone < NumBones; iBone++)
        {
            pFrame = (D3DXBone*)D3DXFrameFind( pRootFrame,
                                pMeshContainer->pSkinInfo->GetBoneName(iBone));

            pMeshContainer->ppBoneMatrixPtrs[iBone] = &pFrame->mtxCombined;
        }
    }

    return S_OK;
}
```

The first thing this function does after casting the mesh container to our derived D3DXMESHCONTAINER_DERIVED structure is test to see if the ID3DXSkinInfo pointer is set to NULL in the mesh container. If not, then this mesh container contains a skin and we retrieve the number of bones that the mesh uses (just as we did in CreateMeshContainer). This value is used to allocate the container's ppBoneMatrixPtrs array to hold that many matrix pointers. Next, we loop through each bone stored in the ID3DXSkinInfo object and retrieve its name so that we can use the D3DXFrameFind function to search the hierarchy (starting at the root) for a frame with a matching name. Once we have a pointer to this frame, we store a pointer to its absolute matrix in the corresponding element in the ppBoneMatrixPtrs array. Remember, these absolute matrices will be rebuilt every time the hierarchy is updated. So this array provides fast access to the matrices after modifications have been made without have to perform a traversal of the tree.

We now have the absolute bone matrix pointers stored in the mesh container in exactly the same order that the bone offset matrices were stored in ID3DXSkinInfo. This provides a one-to-one mapping between pBoneOffsetMatrices and ppBoneMatrixPtrs in the mesh container for easy access later on.

11.7.6 Animating the Skeletal Hierarchy

Skeletal hierarchy animation will be no different from the animation techniques we studied in Chapter Ten. We simply use the animation controller interface to set and activate mixer track animation sets and then call the AdvanceTime method to run the process. After the relative matrices of the hierarchy (i.e. the skeleton) have been updated by the controller, we will build the absolute matrices in the hierarchy by traversing through the structure, combining relative matrices as we go. When our traversal is complete, each bone in the skeleton will have its absolute world space position and orientation stored in its mtxCombined member. If the frame is being used as a bone for a skin, a pointer to this combined matrix will be stored in the mesh container's ppBoneMatrixPtrs array.

The following code shows how a function could be called to update the animation of the bone hierarchy, which is immediately followed by the call to UpdateFrameHierarchy (which traverses the hierarchy and calculates the combined frame matrices).

```
void UpdateAnimation ( D3DXMATRIX matWorld, float m_fElapsedTime )
{
     if ( m_pAnimController != NULL )
          m_pAnimController->AdvanceTime(m_fElapsedTime);

     UpdateFrameMatrices(m_pFrameRoot, &matWorld);
}
```

You can see that there is nothing new here at all. We pass in the world matrix that we will use to position and orient the bone hierarchy and the updated time in seconds that is used to update the animation controller's internal timer. We will actually have a function like this as part of our CActor class so that we can individually update each skinned mesh in our scene. Nevertheless, do not forget that it is possible for multiple skinned meshes to be stored in a single hierarchy. In that case, we would be updating the animation and position of the entire hierarchy stored in a specific actor using the above code.

11.7.7 Transforming and Rendering the Skin

After we have applied skeletal animation and updated the absolute matrix for every bone in the hierarchy, we are ready to traverse the hierarchy and render any meshes that it contains. The hierarchy traversal is no different from the previous lessons -- we simply traverse the hierarchy and render any mesh containers we find (visibility testing preferably included). However, before we render a mesh container, we will want to see if it is a skinned mesh or not. If it is not, then we will render the mesh just as we did before. However if the mesh is a skinned mesh, then we will need to call ID3DXSkinInfo::UpdateSkinnedMesh to transform and blend the vertices from the source mesh into world space and store them in the destination mesh for rendering.

We will not bother showing the code that traverses the hierarchy and searches for mesh containers to render since we covered this in earlier lessons and it remains unchanged. The function we will look at is the one that renders each mesh container, called during hierarchy traversal for every mesh container found in the hierarchy. The function in this example is passed a pointer to the mesh container to be rendered and a pointer to the frame which owns the mesh container. The owner frame is important if this is not a skinned mesh because this frame's combined matrix will be the matrix we wish to use as the world matrix to render the mesh.

The first thing this function should do is check the mesh container's ID3DXSkinInfo interface pointer to see if it is NULL, because this informs the function whether the stored mesh is to be used as a skin. If not, then the mesh can be rendered normally. If it is a skin then additional steps must be taken.

```
void DrawMeshContainer(  LPD3DXMESHCONTAINER pMeshContainerBase ,
                         D3DXFRAME pFrameBase  )
{

    D3DXSkinContainer* pMeshContainer = ( D3DXSkinContainer* ) pMeshContainerBase;
    D3DXBone *pFrame = (D3DXBone*)pFrameBase;

    // first check for skinning
```

```
    if (pMeshContainer->pSkinInfo != NULL){
```

If we get into this code block then the mesh container stores a skin. When this is the case, the first thing we do will be to set the world matrix to an identity matrix, since we will not require the pipeline to transform our vertices into world space. We will do this manually using software vertex blending. We also get the number of bones used by the mesh.

```
            D3DXMATRIX  Identity;
            DWORD          NumBones  = pMeshContainer->pSkinInfo->GetNumBones();
            DWORD          i;
            PBYTE            pbVerticesSrc;
            PBYTE            pbVerticesDest;

            // set world transform
            D3DXMatrixIdentity(&Identity);
            m_pd3dDevice->SetTransform(D3DTS_WORLD, &Identity);
```

Now we need to loop and combine each absolute bone matrix used by this mesh with its corresponding bone offset matrix. We will store the concatenated result in our global temporary matrix array.

```
           // set up bone transforms
            for ( i = 0; i < NumBones; ++i)
            {
                D3DXMatrixMultiply
                (
                    &pgbl_CombinedBoneMatrices[i],
                    &pMeshContainer->pBoneOffsetMatrices[i],
                    pMeshContainer->ppBoneMatrixPtrs[i]
               );
       }
```

We now have an array of matrices to use for transforming the vertices from the source mesh into the destination mesh. Thus, we lock the vertex buffers in both the source and destination meshes for access to the memory.

```
       pMeshContainer->pOriginalMesh->LockVertexBuffer(D3DLOCK_READONLY,
                                                       (LPVOID*)&pbVerticesSrc);
       pMeshContainer->MeshData.pMesh->LockVertexBuffer(0,
                                                       (LPVOID*)&pbVerticesDest);
```

We now pass the pointers for both vertex buffers into the UpdateSkinnedMesh function. We also pass in the combined Bone/BoneOffset matrices we just stored in the temporary global array. When the function returns, the destination mesh in the mesh container will store the new world space blended vertex data for the skin in its current position (based on its current skeleton).

```
     // generate skinned mesh
     pMeshContainer->pSkinInfo->UpdateSkinnedMesh( pgbl_CombinedBoneMatrices, NULL,
                                                   pbVerticesSrc, pbVerticesDest);
```

Note that the second parameter to the above function is supposed to be a pointer to an array of bone offset matrices, but we passed in NULL. As discussed earlier, we decided to combine the matrices

ourselves in order to handle the matrix copy that would have been necessary if we wanted to pass the matrix arrays in separately (because our mesh container stores matrix pointers, not matrices). UpdateSkinnedMesh will recognize that the bone offset array parameter is NULL and will therefore skip the concatenation process that we just performed ourselves.

We can now unlock the two vertex buffers.

```
            pMeshContainer->pOrigMesh->UnlockVertexBuffer();
            pMeshContainer->MeshData.pMesh->UnlockVertexBuffer();
```

At this point, the mesh container now has the updated world space vertex data for the skin stored in the MeshData member (our destination mesh) so we can proceed to render this mesh as we would any other mesh. Below we see a call to a function called DrawMesh, which we can imagine would just loop through each of the mesh subsets, set any relevant textures and materials, and then call the DrawSubset function until the mesh was rendered in its entirety. The destination mesh is just a regular mesh after all and can be rendered using all of the standard techniques.

```
            / / render the mesh
            DrawMesh ( pMeshContainer->MeshData.pMesh )
    }
```

If this mesh container did not store a skinned mesh, then we will just set the world matrix to the combined matrix of its owner frame and render it (as discussed in previous lessons).

```
    else
    {
          IDirect3DDevice9 * pDevice;
          pMeshContainer->MeshData.pMesh->GetDevice(&pDevice);
          pDevice->SetTransform(D3DTS_WORLD, &pFrame->mtxCombined);
          DrawMesh(pMeshContainer->MeshData.pMesh)
          pDevice->Release();

    }  // end if this is not skinned mesh

} // end function
```

11.7.8 Software Skinning Summary

When we break it down into its separate parts, software skinning is really quite simple. We have seen that we can load a skeleton from an X file using the familiar D3DXLoadMeshHierarchyFromX and we now know that the skeleton is simply a standard frame hierarchy, where each frame is considered to be a bone. We also know how to modify our CreateMeshContainer function to support the loading of skinned mesh data from an X file. We saw that this function is passed a standard mesh containing the model space vertices of the mesh in its reference pose and an ID3DXSkinInfo interface which tells us which frames in the hierarchy are used by the mesh as bones. Animation was also quickly taken care of since the skeleton is treated in exactly the same way as any normal frame hierarchy.

Finally, we saw that skinned mesh rendering requires a few additional steps, but is also straightforward enough. We use the ID3DXSkinInfo::UpdateSkinnedMesh function to convert the model space skin vertex data into world space vertex data using vertex blending. We pass this function an array of bone and bone offset matrices along with our source mesh vertices and a destination vertex buffer that will receive the transformation results. The ID3DXSkinInfo object will manage the entire transformation process using the bone weights stored internally. It can do this because the ID3DXSkinInfo object contains all the vertex weights for each influential bone that was stored in the X file. As discussed earlier, these weights are used to scale the contribution of bones on one of its assigned vertices.

Once we have the world space vertex data in the destination mesh, we can simply set the world matrix of the device to an identity matrix and draw the mesh subsets as we normally would.

In the workbook that accompanies this chapter you will see that code similar to the sample code shown in this textbook has been implemented inside our CActor class. We will implement both software skinning and two different types of hardware skinning techniques in the code, so be sure to focus on the software skinning technique for now. We will begin to tackle the hardware techniques in the next section.

Software Skinning Advantages

- Very easy to understand and implement.
- Requires no hardware support and will therefore work on any system.
- There is no limit to how many bones a single vertex can be attached to because all of the bone/weight information is stored in the ID3DXSkinInfo object. The only limits we face are more practical: the memory taken up by the bone/vertex mapping information and the speed at which the vertices can be transformed by the CPU.
- Because the transformation is done in software, software skinning does not affect the way we batch render our primitives. This allows us to continue to batch by states such as textures and materials for optimal performance. This is not true for hardware skinning; we will be forced to batch by bone contributions as the primary key (discussed later).

Software Skinning Disadvantages

- Often considerably slower than hardware skinning techniques because vertices are transformed in software inside the UpdateSkinnedMesh method of the ID3DXSkinInfo interface. With dedicated hardware skinning, the vertices are multi-matrix transformed in the DirectX pipeline (by the GPU on a T&L card). As we will see however, non-indexed hardware skinning (unfortunately, the more commonly supported hardware skinning technique) can often suffer a performance hit as well due to less than ideal batching requirements.

11.8 DirectX Skinning II: Non-Indexed Skinning

The DirectX fixed-function pipeline provides two skinning methods that can take advantage of available 3D hardware support for vertex blending. It is worthy of note that the software skinning technique we have just discussed is not one of these techniques, since it was not performed in the DirectX pipeline. Software skinning simply sent a regular mesh to be transformed and rendered by the pipeline in the usual way; the fact that this mesh was being used as a skin was of no significance to the pipeline. This was because we transformed the vertices into world space manually with the help of the ID3DXSkinInfo interface. When it came time to render the skin, we simply handed it off to the pipeline as a regular mesh (albeit containing world space vertices). The pipeline was not aware that the vertices were transformed according to some skeletal pose or how they ended up in the destination mesh. So it is very important to remember, as we move forward, that the software skinning technique previously discussed was *not* performed in the T&L pipeline.

There are two geometry blending techniques that are integral to the pipeline. They can be invoked as an alternative to software skinning to accelerate the skinning process in hardware. These techniques are referred to as **non-indexed skinning** and **indexed skinning**. The latter is the more efficient of the two, but support is not as widely available, especially on older T&L graphics hardware. However, even if hardware support is not available for either of these techniques, we can still use the pipeline's skinning techniques by rendering the mesh using software vertex processing. This will of course mean that the vertex blending is carried out in the software module of the vertex transformation pipeline and thus is not going to be as fast. But the important point is that both techniques will work on all systems even when no hardware support is available. The software vertex blending module of the DirectX pipeline is still quite efficient, so this is a good fallback for us even in commercial applications.

You should note the distinction between software skinning as discussed previously and the pipeline skinning techniques performed using software vertex processing. As mentioned, in software skinning the pipeline plays no part in the generation of the world space vertex transformations for the skin. But when using either indexed or non-indexed pipeline skinning, we do not have to transform the vertices of the skin from model to world space ourselves. We simply pass the skin to the pipeline as model space vertex data and the pipeline will multi-matrix blend the vertices into world space for us. If hardware support is available, then the GPU will be utilized to perform this geometry blending with great speed and efficiency. If hardware support is not available, the pipeline can still perform the blending transformations for us using its own software transformation module.

If we intend to utilize the pipeline's vertex blending module on a system that does not support hardware acceleration, we will need to place the device into software vertex processing mode prior to rendering the skin so that the software module can be utilized by the pipeline. This also means that the mesh that contains the skin will need to be created with the D3DXMESH_SOFTWAREPROCESSING flag so that the software module of the pipeline can be allowed to work with its vertices. This brings other considerations to the fore. We know from our discussions in Chapter Three that in order for the pipeline to be able to perform software vertex processing, we must either create a software device or a mixed-mode device. This contrasts with the kind of device we would usually be striving for -- either a hardware device or the even more efficient PURE device. We cannot use software vertex processing with the latter two device types, so it should be obvious that if we intend to use the pipeline skinning techniques

and we want our application to work on an end user system that does not have hardware support for vertex blending, ideally we will want to create a mixed-mode device. This will allow us to render of all our regular geometry with the device set to hardware processing mode (so that the vertex data is transformed and lit by the GPU) and then switch to software processing mode for our chosen pipeline skinning technique. Although the default state of a mixed-mode device is hardware vertex processing mode, if the device had previously been placed into software vertex processing mode, we would want to place it back into hardware mode prior to rendering our regular meshes for maximum speed. To do so we would use the following function call:

```
pDevice->SetSoftwareVertexProcessing(FALSE);
```

When the time comes to render our skinned meshes (and no hardware support exists), we can place the mixed mode device into software mode as follows:

```
pDevice->SetSoftwareVertexProcessing(TRUE);
```

Anything we render from this point on will use the DirectX software transformation pipeline. While performance will not be as good, we have given ourselves the option of using the pipeline skinning techniques even when hardware support for vertex blending is absent on the current system

Of course, in order to switch between hardware and software vertex processing techniques you will need to have created a mixed-mode device. If you had created a hardware device or a PURE device, then software vertex processing could not be invoked and the pipeline would fail to render the skin on a device that had no hardware support.

> **Note:** Be careful not to set the vertex processing mode to software for all of your rendering (even on a mixed-mode device). Once software vertex processing is enabled, any T&L acceleration that the 3D card has to offer will be ignored and all vertices rendered from that point on (until software vertex processing is disabled) will be transformed and lit in software. This will incur a significant performance penalty if you use it to render your entire scene rather than just the meshes that require software emulation to account for the lack of hardware support.

11.8.1 Why use software skinning at all?

If the DirectX pipeline can perform the vertex blending in software when no hardware support is available, then why would we ever need to use the software skinning technique we discussed in the previous section? The answer will become clearer as we discuss the semantics of the DirectX transformation pipeline with respect to vertex blending, but for starters, we know that with software skinning, we are not limited by how many bones are allowed to influence a single vertex. This is because the weights for each vertex for a given bone are stored inside the ID3DXSkinInfo object and are used when we manually transform the mesh using UpdateSkinnedMesh. However, when using the pipeline to perform vertex blending, we do not have the luxury of storing elements such as vertex weights in external data objects like ID3DXSkinInfo. Once we call DrawPrimitive, the model space vertices of the skin are sent off into the pipeline and from that point forward things are pretty much out of our hands. When the pipeline is performing vertex blending, we have to pass the weights in the vertex

structure itself. This is how the pipeline (and hopefully the GPU, if hardware support is available) is informed as to how strongly each currently set world matrix is to influence the vertex being transformed.

In a few moments we will look at how to store the weight values in the mesh vertices as well as how to set multiple world matrices (bone matrices) on the device. The important point right now is that there is a limit on how many weights can be stored in a vertex. This is because vertices need to be compact packets of information that can be passed over the bus at a reliably decent speed and stored efficiently in either local or non-local video memory. Therefore, DirectX limits the number of weights that we can store and use in a vertex to 3 in the non-indexed skinning technique. This will mean that a single vertex can, at most, be influenced by four matrices. Why four? As it happens, the pipeline expects the combined weights stored in the vertices to always add up to 1.0, so if we have three weights stored in a vertex, the pipeline can calculate the fourth weight by summing the first three and subtracting from 1.0. This clever design decreases the memory footprint of vertices in precious video memory and lowers the amount of subsequent bus traffic.

Weight4 = 1.0 – (Weight1 + Weight2 + Weight3)

Although four weights per vertex might not sound like much, it is generally more than adequate given how most skinned meshes are constructed and animated. Unfortunately, the situation is quite a good deal worse when we use non-indexed vertex blending. You will see in a moment how we can set up to four world (i.e. bone) matrices on the device. That is, before a vertex is sent to the pipeline, we will need to set the bone matrices that influence it. The weights in the vertices match a corresponding matrix; the pipeline will use the first weight defined in the vertex to weight the contribution of the first currently set world matrix, the second weight will be used to weight the contribution of the second currently set world matrix, and so on.

However, we know that the vertex blending pipeline is invoked after we have called the DrawPrimitive function (or the DrawSubset wrapper function), so the smallest entity we will be rendering will be at least one triangle. We do not render one vertex at a time, but rather one triangle at a time, so it is only between DrawPrimitive calls that we can change states -- like setting different world matrices. As a result, when using non-indexed blending, not only are we are limited to a maximum of four matrices per vertex, but consequently, we are limited to four matrices *per subset*.

This last point has a real impact on our ability to batch render effectively. Since we can only change matrices between draw calls, our ability to batch render triangles has been greatly reduced. We will now need to attribute sort our mesh not only by textures and materials, but primarily by bone (matrix) contributions. This implies re-ordering the mesh triangles such that all triangles that use the same four bones will belong to the same subset. This can greatly reduce the number of triangles that can exist in a single subset and therefore greatly reduce the number of triangles in the skinned mesh that we can render with a single call to DrawPrimitive. For example, a typical non-skin mesh might have 20 triangles that use the same texture and material and usually we would attribute sort the mesh such that all of these triangles would belong to a single subset. However, in a skin mesh, each one of these triangles might be attached to a different combination of 4 bones. While this is probably an extreme example, the point is that it would cause not one, but twenty subsets to be created, each with a single triangle and each requiring its own DrawPrimitive call. Again, we have to batch by bone because we will need to set the bone matrices on the device before we render a given subset.

As if things do not sound bad enough, three weights (four matrices) is the maximum that the DirectX pipeline supports, so we will be able to use all three weights if we are using software vertex processing. However, if we want the pipeline to take advantage of hardware support for vertex blending, then we may hit another wall -- the graphics hardware might not support all four matrix blends. In some cases only two or three matrix blends are supported. When this is the case, the mesh will have to be modified so that only the supported number of weights is used in each vertex. This would usually involve having to step through the weight/bone information, removing less influential bones from attached vertices. The result is that the skin might not animate as nicely as the artist had intended. If a given vertex was assigned to four bones by the artist, but the current hardware only supports two bones, two bones would need to be detached from this vertex and not used. The vertex would now store only one weight value to blend the two matrices we would use to transform it.

You may be thinking that the thought of loading a mesh and determining how many bones the hardware supports and having to convert that data into a hardware friendly mesh, batched into subsets based on bone contributions, sounds like a complete nightmare. However, you will be glad to know that support is available. When the mesh is first loaded inside the CreateMeshContainer function, we can use the ID3DXSkinInfo::ConvertToBlendedMesh function to perform all of these steps for us. We will look at this function in more detail later, but for now just know that we will pass this function the regular ID3DXMesh containing the skin that is passed into our CreateMeshContainer function. ConvertToBlendedMesh will automatically clone the mesh into a format that contains the correct number of vertex weights in each vertex and will attribute sort the mesh so that triangles are batched primarily by bone combinations. This means that all of the triangles that use one set of up to four matrices will belong in one subset; all the triangles that use another combination of four matrices will belong in another subset; and so on. Subsets are then further subdivided so that a single subset will contain only triangles that share the same material and texture. As you can imagine, this means we can end up with subsets that are quite small, even in typical cases.

For example, the skinned mesh contained in ‘Tiny.x’ which ships with the DirectX SDK uses only a single texture and material. This is something we try to target when working with skinned meshes, for obvious reasons. If we were to register Tiny as a regular mesh, it would have only one subset and we could render it with a single draw call. However, when Tiny is converted into a skin that is compatible with the non-indexed skinning technique, the new mesh will have been broken up into over 30 different subsets (and that is assuming that four matrix blends are available on the hardware). So even though each subset uses the same texture and material, the triangles in each subset use different combinations of 4 bone matrices which need to be set on the device before we issue each draw call. This batching problems stems not from only being able to have four bone influences per vertex, but from the implied one-to-one mapping between vertex weights and matrix slots on the device. As there are at most four weights, they always map to the first four matrices set on the device.

We can check how many blend weights the mesh returned from the ConvertToBlendedMesh function needs to use and make some decisions of our own. If this value is beyond the limits of the hardware, we can choose to invoke software vertex processing at this point (device type permitting).

When it comes time to render the mesh, we will loop through each subset, set the world (bone) matrices that the triangles in that subset are influenced by (which ID3DXSkinInfo tells us), and then render that

subset with vertex blending enabled, using the D3DRS_VERTEXBLEND renderstate that we will take a look at shortly.

In this section we will discuss non-indexed blending. This is the skinning technique which is most commonly supported by hardware (even on older T&L cards) but has the disadvantages just described. Later in this lesson we will discuss the second hardware accelerated skinning technique called indexed blending, where we are no longer limited to only four influences per subset. This will allow for much more efficient batching.

11.8.2 Setting Multiple World Matrices

In order for the DirectX pipeline to do a multi-matrix blend of the vertices in our skinned mesh, we need a way to bind more than one world matrix to the device at a time. When using non-indexed blending, a vertex can have up to three weights, so it will require the pipeline to transform it using up to four world matrices. Whereas in the software skinning case we simply passed our entire array of bone matrices into the UpdateSkinnedMesh function, now we will need to send the device only the bone matrices that the subset we are about to render will use.

As it turns out, the software skinning code we used to extract the bone offset matrices from ID3DXSkinInfo and store them in the mesh container can be reused. This part of the system will not change -- we will continue to use the ID3DXSKinInfo object passed into the CreateMeshContainer function to retrieve the bone offset matrices and store them in the mesh container. We will similarly traverse the hierarchy to find the matching absolute bone matrices so that we can store those in the mesh container as well. Therefore, when it comes time to render, our mesh container will store an array of bone offset matrices and a matching array of absolute bone matrix pointers just as it did in the software skinning case.

The difference now is that we will not simply combine all bone matrices with their corresponding bone offset matrices into the global matrix buffer and run the update (although we will still need to perform the concatenation). Instead, we will first need to determine which bones in our bone array are needed by the subset we are about to render. This is because these are the bones that the weights defined in the subset vertices are relative to. When we examine our modified CreateMeshContainer function a bit later in the lesson, we will see how to convert the skin into a mesh that contains the vertex weights. We will be able to subsequently retrieve the mesh in the desired vertex format along with something called a *bone combination table*. The bone combination table will be an array of structures, where each element in the array will contain the bone indices used by a given subset. Before we render skin subset N, we will check element N in our bone combination table to determine which bones are relevant. This structure in the table will store an array of bones indices which we can use to locate the needed world matrices in the mesh container. Because we stored our bone matrix pointers inside the mesh container in the same order that the bone offset matrices were stored in the ID3DXSkinInfo object, these bone combination table indices will directly index into our array of bone matrix pointers and bone offset matrices stored in the mesh container.

As an example, let us imagine that we are about to render subset 5 of a mesh, and element 5 in our bone combination table tells us that this subset uses bones 6 and 20. We know that we must combine each absolute bone matrix with its bone offset matrix stored in the corresponding mesh container arrays before we set them on the device as a world transformation matrix:

```
D3DXMatrixMultiply(&Matrix1,
                   &pMeshContainer->pBoneOffsetMatrices[6],
                    pMeshContainer->ppBoneMatrixPtrs[6] );

D3DXMatrixMultiply(&Matrix2,
                   &pMeshContainer->pBoneOffsetMatrices[20],
                    pMeshContainer->ppBoneMatrixPtrs[20] );
```

As you can see, the two bones that influence the subset we are about to render have been combined with their corresponding bone offset matrices such that we now have two temporary combined matrices which we can bind to the device as world matrix 1 and world matrix 2. For each vertex in this subset, its first weight will determine how influential Matrix1 (bone 6) will be and the second weight will determine how influential Matrix2 (bone 20) will be.

For the time being, do not worry about how we find out which bones affect which subset; we will see this all later when we take a detailed look at the ConvertToBlendedMesh function. The bigger point here is that in the above example we have created two combined matrices which we now need to set on the device so that the pipeline can access them. So how do we set more than one world matrix on the device?

The World Matrix Palette

A Direct3D device actually has the ability to store 511 matrices. When we call the SetTransform device method and pass in a transform state such as D3DTS_VIEW to set the view matrix, this transform state actually equates to an index into a larger palette of matrices which places the view matrix at the index in this matrix palette where the device expects it to be stored. The first 256 matrix positions (transform states 0 – 255) are used by the pipeline to store the view matrix, the projection matrix, the viewport matrix, and other matrices such as the texture matrices for each texture stage. We will not normally place matrices into these positions using raw matrix indices. Rather, we use transform states such as D3DTS_VIEW and D3DTS_PROJECTION to map to matrix palette indices somewhere in the 0 – 255 range. However, transform states 256 – 511 are reserved to hold up to 256 world matrices. Up until now, we have only used one of these world matrices so you might have assumed that there was only one world matrix slot on the device. As it happens, when we set a world matrix like so:

```
pDevice->SetTransform ( D3DTS_WORLD , &mat );
```

we are actually setting this matrix in the device matrix palette at index 256 (the first palette position where world matrices begin). That is, D3DTS_WORLD is defined in the DirectX header files with a value of 256. When we are not using vertex blending, we only need to use this first world matrix position. But we now see that we have the ability to set many more world matrices simply by passing in

the index of the matrix we wish to set. For example, if we wanted to set three world matrices, we could do the following:

```
pDevice->SetTransform ( 256 , &mat1 );
pDevice->SetTransform ( 257 , &mat2 );
pDevice->SetTransform ( 258 , &mat3 );
```

Bear in mind however that setting the matrices does not mean that the transformation pipeline will use them -- we have to enable vertex blending for that to happen. If the vertex blending render state has not been set to TRUE, then only the first world matrix (256 or D3DTS_WORLD) will be used to transform vertices. Since it is not very intuitive to set the world matrices using raw index numbers, DirectX supplies us with a macro that maps indices 0-255 into the 256-511 range. This is useful because if we know we have to set four bone matrices to render a given subset, we tend to think of these as being matrices 0-4 and not matrices 256-259. Therefore, when setting world matrices we will use the D3DTS_WORLDMATRIX macro, defined in the DirectX header files as:

```
#define D3DTS_WORLDMATRIX(index) (D3DTRANSFORMSTATETYPE)(index + 256)
```

As you can see, this simple macro just adds 256 to the passed index. Thus, we would rewrite the above example code as:

```
pDevice->SetTransform ( D3DTS_WORLDMATRIX ( 0 ) , &mat1 );
pDevice->SetTransform ( D3DTS_WORLDMATRIX ( 1 ) , &mat2 );
pDevice->SetTransform ( D3DTS_WORLDMATRIX ( 2 ) , &mat3 );
```

Although the result is the same, it is much more intuitive to read. We can see that matrix (0) maps to weight (0) in the vertex, matrix (1) maps to weight (1) in the vertex, and so on. In this example where only three matrices are being used, you should keep in mind that the weight for matrix (2) would not be stored in the vertex since the pipeline will automatically calculate it by doing 1.0 – (weight0 + weight1). If a maximum of N matrices are being used, there should be N-1 weights in the vertex structure. Weight N is always calculated by the pipeline.

You may be wondering why 256 world matrix slots have been allocated when we know that a maximum of four matrices is supported. After all, in the current version of DirectX, only three weights can be defined in the FVF format of the vertex (the fourth weight is calculated on the fly) and these are paired up with the corresponding matrices that are set on the device. Since there is an implicit mapping between weight positions in the vertex and matrix indices on the device, it must follow then that given the restriction of four weights, only four of the possible 256 matrices will ever be used during blending. The other 252 matrices would appear to be wasted space. As it turns out, when performing non-indexed blending, this is absolutely the case. At most, we will only ever use the first four world matrix slots on the device (Fig 11.20). It is because of this limitation that the non-indexed skinning technique suffers such inefficient batching (all of the triangles in a given subset must use the same four bone matrices in order to be rendered together).

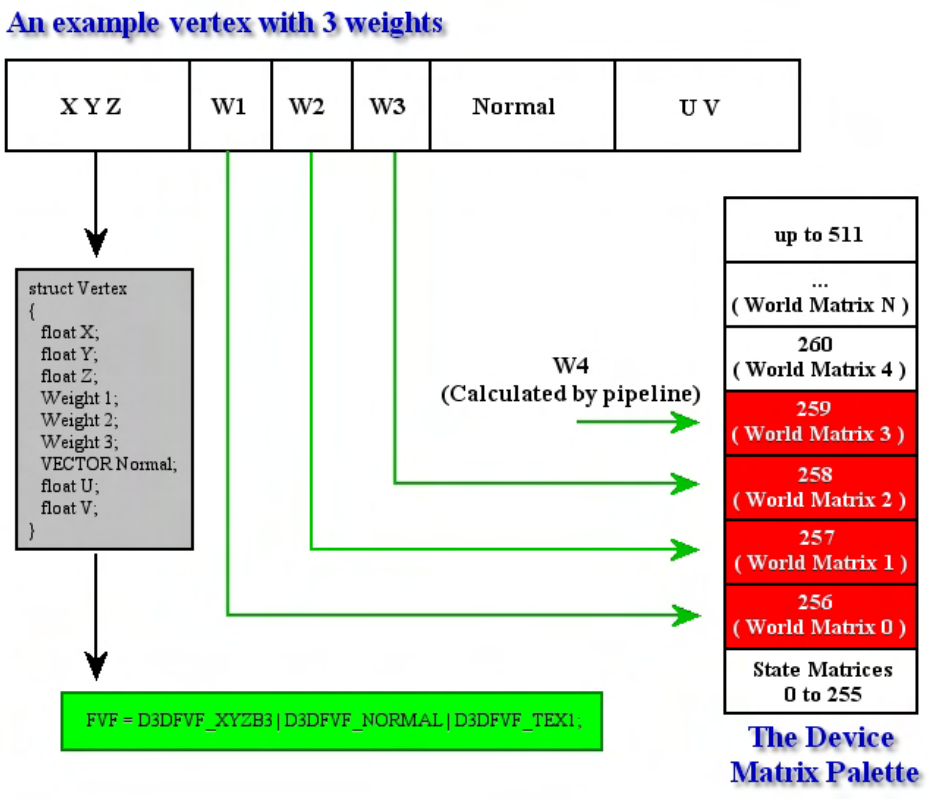


Figure 11.20

In Fig 11.20 we see an example of a vertex format that contains three weights, which gives us access to the first four world matrices on the device. Notice the FVF flags in the green box that we use to define the vertex format for the vertex buffer that would contain these weighted vertices. The D3DFVF_XYZB3 flag is an FVF flag we have not encountered yet in this series. It is used in place of the typical D3DFVF_XYZ to specify that each vertex will have three blend weights defined. As you might imagine, there are other flags for defining vertices with different blending weights (one, two, three, four, or five weights as it turns out):

Additional FVF blending flags:
D3DFVF_XYZB1
D3DFVF_XYZB2
D3DFVF_XYZB3
D3DFVF_XYZB4 // not currently supported
D3DFVF_XYZB5 // not currently supported

These FVF flags are mutually exclusive since they each define a vertex as having untransformed X, Y, and Z components along with some specified number of weights. Fig 11.20 also shows us that the weights should be defined immediately after the XYZ position component in the vertex structure. The number of float weights you place in your vertex must match the FVF flags that you intend to use.

We will not actually need to create a vertex buffer ourselves using the flags listed above because the ConvertToBlendedMesh function will create the mesh (and therefore its vertex buffer) for us in the correct format. However, you are certainly permitted to use the pipeline's vertex blending features

without having used this particular function to set up the mesh on your behalf. In such cases you would need to create the vertex buffer yourself with the appropriate FVF flags and vertex weight values.

Later in the lesson we will discuss the second pipeline skinning technique called indexed blending. It is much more efficient, but unfortunately not as widely supported in hardware. Using this technique, each vertex can contain up to 3 weights as in the non-indexed case, but can also contain 4 indices using a packed DWORD. These indices allow the vertex to specify not only the weights themselves, but exactly which bone matrix in the device matrix palette that each weight applies to -- up to 256 now allowed. Consequently, subsets will be much larger and even a single triangle could be influenced by 12 unique bone matrices. Since a subset could contain many triangles all influenced by different bones somewhere in the 256 matrix palette, this makes a significant difference. We mentioned the case of Tiny.x earlier and saw that over 30 skin subsets were created when using non-indexed blending (each of which could only access four matrices at a time). In the indexed case, the mesh contains only a single subset and can be rendered with a single draw call. This is because we can set all of the bone matrices on the device for the character simultaneously before we render the mesh. The vertices of the mesh accurately describe to the pipeline (using their stored indices), which of the currently set matrices its weights are defined for.

For the time being, let us not get distracted by indexed vertex blending, for we still have a ways to go in our examination of non-indexed blending. Like it or not, non-indexed blending will often be the only hardware accelerated vertex blending technique supported on an end user's system, so you will definitely want to accommodate it in your skinned mesh implementation.

11.8.3 Enabling Vertex Blending

To enable vertex blending on the device we use the D3DRS_VERTEXBLEND render state:

```
pDevice->SetRenderState ( D3DRS_VERTEXBLEND , D3DVERTEXBLENDFLAGS );
```

The second parameter should be a member of the D3DVERTEXBLENDFLAGS enumerated type. The default state for this render state is D3DVBF_DISABLE, which means the pipeline will perform no vertex blending and a single world matrix (stored at index 256) will be used for vertex transformations. This is the mode we have used in all our demos up until this point where a single world matrix was needed.

```
typedef enum
_D3DVERTEXBLENDFLAGS
{
        D3DVBF_DISABLE = 0,
        D3DVBF_1WEIGHTS = 1,
        D3DVBF_2WEIGHTS = 2,
        D3DVBF_3WEIGHTS = 3,
        D3DVBF_TWEENING = 255,
        D3DVBF_0WEIGHTS = 256
} D3DVERTEXBLENDFLAGS;
```

Before rendering the mesh, we will tell the device how many matrices we wish to use per vertex. One point that needs to be clear is that if we pass D3DVBF_2WEIGHTS for example, then the vertices in our mesh must have *at least* two weight components defined in the FVF. However, keep in mind that every vertex in the mesh must have the same number of weights because all vertices in a mesh vertex buffer must be of the same format.

We will use the ConvertToBlendedMesh function to convert our initial skin mesh into one that has the correct number of weights. This function essentially informs the pipeline of the number of weights defined in each vertex, which will be equal to the maximum number of bone influences for a single vertex in the mesh (minus 1 because the last weight is never stored, it is calculated on-the-fly).

If our mesh had all of its vertices influenced by only two bones, but one vertex in the mesh happened to be influenced by four bones, then every vertex in the mesh would have three weights defined and we would use the D3DVBF_3WEIGHTS flag before rendering this mesh. Any vertices that are only influenced by two bones (and would normally only need one weight) will still have three weights -- the two additional weights are added to pad the vertex structure. They will be assigned weights of 0.0 so that they do not affect the vertex transformation.

Although these 'padding' weights will not affect the final transformation results for the vertex, it is necessary to note that they will still be used to blend matrices, even though the result will be a no-op because it is scaled to zero. We will see later how we can introduce a possible optimization during rendering in the non-indexed case. Essentially, before rendering a subset, we will find out the maximum number of matrices used by that subset and adjust the D3DRS_VERTEXBLEND render state to this amount. This will allow us to make sure that the additional (padding) weights are not used to needlessly blend matrices that will have no effect on the final vertex position. Using our previous example, where the mesh vertices have been padded to three weights and most of the subsets are influenced by only two matrices, we would determine before we render a subset which of the two categories it falls into. If it is influenced by two matrices then we can set the D3DRS_VERTEXBLEND to D3DVBF_1WEIGHTS so that the additional padded weights (weights 2 and 3 in this example) and their respective matrices are not needlessly included in the vertex transformation.

Therefore, the second parameter to this function describes not the total number of weights defined in the vertices, but the number we wish to use for transformations (which may be less). Let us examine the possible values the D3DRS_VERTEXBLEND render state can be set to and see what they mean when used.

D3DVBF_0WEIGHTS

This value informs the device that the vertices we are about to render do not contain any weights and therefore will be transformed by only one matrix with an assumed weight of 1.0. This might sound like a very strange flag to be able to set because surely a vertex that uses one matrix with a weight of 1.0 is a vertex that is not being blended. Indeed it would seem that this flag would have the same effect as using the D3DVBF_DISABLE flag. As it turns out, this is actually true in the case of non-indexed blending and in fact, there will be no need to use this flag under those circumstances. However, in the indexed blending case, we might have a triangle where each of its three vertices will need to use different world matrices. Therefore, while vertex blending is not taking place, it is still possible that we might have a mesh where every vertex is only attached to one bone, but the vertices belonging to the same triangles/subsets might not all use that same bone for transformation. While no actual blending is taking place, three matrices could still be used to transform a single triangle (e.g., one matrix for each vertex). This would not be possible using only the single world matrix that is available using the D3DVBF_DISABLE flag. This will all make more sense when we discuss indexed skinning later in the lesson.

D3DVBF_1WEIGHTS
This flag informs the device that we wish to render our triangles using vertex blending where each vertex will have a single weight. Remember that if the vertex has one weight, then this means it is influenced by two bones and will be blended using two matrices. The second weight is calculated by the pipeline as Weight2 = 1.0 – Weight1. After setting this state, any triangles that you render must have at least one vertex weight defined or behavior will be undefined.

D3DVBF_2WEIGHTS
This flag informs the pipeline that we wish to render our triangles such that they will be transformed using three world matrices. After this state has been set, any triangles we render must have a vertex format that contains at least two weights. When this state is set the vertex will be transformed using the first three world matrices on the device. The weight of the third matrix is calculated as Weight3 = 1.0 – (Weight1 + Weight2).

D3DVBF_3WEIGHTS
This flag informs the pipeline that we wish to render some triangles such that they will be transformed using four world matrices. After this state has been set, any triangles we render must have a vertex format that contains at least three weights. When this state is set, the vertex will be transform using the first four world matrices on the device. The weight of the fourth matrix is calculated by the pipeline using the calculation: Weight3 = 1.0 – (Weight1 + Weight2 + Weight3).

D3DVBF_TWEENING
We will not be using vertex tweening in this lesson and therefore we will not require this flag. However you already have a good high level understanding about how tweening works, so you should be able to use the SDK documentation to fill in the gaps if you would like to use this technique in your applications. (Recall that tweening was an interpolation technique between two pre-defined sets of mesh vertices in different key animation poses.)

D3DVBF_DISABLE
This flag will disable vertex blending and set the device back to its default mode of using a single world matrix to transform vertices.

We have now covered all of the theory involved in the non-indexed skinning technique, so it is time to discuss some implementation details. We will need to examine what our CreateMeshContainer function will now look like for the non-indexed skinning case. We will also want to have a discussion about the very important ID3DXSkinInfo::ConvertToBlendedMesh function, since it will do so much of the setup work for us.

11.8.4 Storing the Skinned Mesh

Whether we are using software skinning, non-indexed skinning, or indexed skinning, our application loads the skinned mesh and bone hierarchy in exactly the same way: using the D3DXLoadMeshHierarchyFromX function. It is in the CreateMeshContainer function of our ID3DXAllocateHierarchy derived class that we are passed a regular mesh and an ID3DXSkinInfo interface. Our job will be to use the tools available to create a mesh that is capable of being vertex

blended by the pipeline. Therefore, we will skip straight to the CreateMeshContainer function and the mesh container structure we will need to store the required information.

Our mesh container will need a few additional members when we perform skinning in the pipeline. Also, when performing indexed or non-indexed skinning we will no longer need our temporary global matrix array as we did when performing software skinning. This is because this array was used to store the resulting matrices of combining the bone matrices with their corresponding bone offset matrices before sending them to UpdateSkinnedMesh. We will no longer need this temporary matrix buffer because we will be setting the matrices (a maximum of a four at a time in the current case) directly on the device. The new mesh container will accommodate the CreateMeshContainer function for a non-indexed skinned mesh and is defined next.

```
struct D3DXMESHCONTAINER_DERIVED : public D3DXMESHCONTAINER

{
    DWORD           NumAttributeGroups;
    DWORD           NumInfl;
    LPD3DXBUFFER    pBoneCombinationBuf;
    D3DXMATRIX**    ppBoneMatrixPtrs;
    D3DXMATRIX*     pBoneOffsetMatrices;
    DWORD           iAttributeSW;
};
```

Four new data members have been added to this structure and one has been removed. We no longer need to store the original mesh since the concept of a source and destination mesh is no longer valid. We will now store a single model space mesh in the MeshData member of the base structure and it is this mesh that will be transformed and rendered by the pipeline. Notice that the two matrix arrays remain. As before, the ppBoneMatrixPtrs array will contain pointers to all of the absolute bone matrices in the hierarchy that affect this mesh and we also maintain the corresponding bone offset matrix array. Most of the new members are used to store information that is returned from the ConvertToBlendedMesh function, which we will look at shortly.

DWORD NumAttributeGroups

When ID3DXSkinInfo::ConvertToBlendedMesh is called, our mesh will be cloned into a new format where the vertices contain the correct number of weights. But this function also does a lot more than simply clone the mesh into a different vertex format. Most importantly, it also breaks the mesh into different subsets so that no individual subset is influenced by any more than four matrices (or perhaps less depending on the hardware), since this is all we have available with the non-indexed skinning technique. This is likely to create many more subsets/attribute groups than the original mesh had and we will need to know how many subsets the new mesh has been divided into because there will be exactly this many elements in our bone combination buffer (also returned from the ConvertToBlendedMesh function and stored in the mesh container). Each element in the bone combination buffer will represent a subset in the mesh and will contain information such as the matrix indices into our bone/bone offset arrays which we will need in order to render that subset. We will look at the bone contribution table in a moment.

The NumAttributeGroups DWORD describes how many subsets the blended mesh has and is the number we will need to loop against in order to set the matrices for each subset and render it.

DWORD NumInfl

This member will store a value that will be returned from ConvertToBlendedMesh that describes the maximum number of influences that a single vertex has in the resulting mesh. Subtracting one from this value will tell us exactly how many weights we have in the vertex structure of the converted mesh.

If this value is set to 4 then it means that there is at least one vertex in the mesh that is influenced by four matrices (and will therefore need to have 3 weights stored for it). In that case, every vertex in the mesh will have three weights. Any vertices that are influenced by fewer than 3 weights will have padding weights set to 0.0 as needed.

LPD3DXBUFFER pBoneCombinationBuf

This member stores a pointer to an ID3DXBuffer interface. The interface to this buffer is returned from the ConvertToBlendedMesh function and is stored in the mesh container for use during rendering. The buffer will contain the bone combination table for this mesh -- an array of D3DXBONECOMBINATION structures. Each structure in the array contains information for a single subset in the converted mesh. This means that the array will contain NumAttributeGroups D3DXBONECOMBINATION structures.

When rendering the mesh, we will loop through each of its subsets/attribute groups and fetch its D3DXBONECOMBINATION structure from this buffer. This structure will tell us which texture and material to use, which vertex and index ranges in the vertex and index buffers belong to this subset, and of course, which bone matrices need to be set on the device.

The D3DXBONECOMBINATION structure is defined as follows:

```
typedef struct _D3DXBONECOMBINATION
{
    DWORD AttribId;
    DWORD FaceStart;
    DWORD FaceCount;
    DWORD VertexStart;
    DWORD VertexCount;
    DWORD *BoneId;
} D3DXBONECOMBINATION, *LPD3DXBONECOMBINATION;
```

This is a very important structure for a number of reasons. First, it is our only link between subsets in the converted mesh and their relationship to the bones in our mesh container. It is also our only link between the textures and materials that these new subsets use. Remember that although we are passed the D3DXMATERIAL buffer (in CreateMeshContainer) which describes the texture and material to use for each subset in the original mesh, after we have converted the mesh using the ConvertToBlendedMesh function, the subsets will have been completely rearranged and subdivided.

> **Note:** The bone combination buffer is calculated automatically during ConvertToBlendedMesh, so we will not have to figure out these values ourselves.

Before continuing our mesh container examination, let us take some time to discuss the members of this new structure. Keep in mind that there will be one of these structures in the buffer for each subset in the converted mesh.

DWORD AttribId
This value is our means of indexing into the original D3DXMATERIAL buffer passed into the CreateMeshContainer function. It will describe the texture and material that should be used to render this subset. Remember that the converted mesh will have had its subsets completely recalculated, so there is no longer have a 1:1 mapping with the D3DXMATERIAL buffer. That is why this link between the new subset and old D3DXMATERIAL array is needed.

DWORD FaceStart
DWORD FaceCount
These two members describe the range of triangles in the index buffer of the converted mesh which are considered to be part of this subset. When the ConvertToBlendedMesh function executes, each new subset it generates will have its triangles stored together in the index buffer. FaceStart describes the index of the first triangle in the index buffer that belongs to this subset and FaceCount describes how many triangles (starting at FaceStart) are included in this subset.

DWORD VertexStart
WORD VertexCount
These two members describe the range of vertices in the vertex buffer that belong to this subset. When the ConvertToBlendedMesh function executes, each new subset it generates will have its vertices stored together in the vertex buffer. VertexStart describes the index of the first vertex in the vertex buffer that belongs to triangles in this subset. VertexCount describes how many vertices (starting at VertexStart) are used by triangles in this subset.

DWORD *BoneId
This member is an array of DWORD indices describing which matrices in our mesh container matrix arrays are used by this subset. This tells us which matrices need to be set on the device in order to render this subset. The size of this array will always be equal to the maximum number of influences on a single vertex in the mesh -- which is equal to the NumInfl value we added to our mesh container. As an example, let us imagine once again the case where a mesh might have been converted such that each vertex has three weights. This means that, at most, a vertex will be blended using four matrices. In this instance, the BoneId array of every subset will have four elements in it. These four elements are the indices of the four matrices that the vertices in this subset need to have sent to the device before they are rendered. If the current subset being processed only uses two matrices out of the possible four, then the last two bone indices in the subsets BoneId array will be set to UINT_MAX and their corresponding weights in each vertex will be set to zero.

To better illustrate what BoneId contains, let us imagine that the current subset we wish to render has a BoneId array containing the values {4, 3, 1, UINT_MAX}. Since the vertices in this subset are influenced by only three of the possible four matrices we can set, we would need to set the following matrices on the device before rendering this subset:

```
D3DXMatrixMultiply(&Matrix0,
                   &pMeshContainer->pBoneOffsetMatrices[4],
                   pMeshContainer->ppBoneMatrixPtrs[4] );

D3DXMatrixMultiply(&Matrix1,
                   &pMeshContainer->pBoneOffsetMatrices[3],
                   pMeshContainer->ppBoneMatrixPtrs[3] );
```

```
D3DXMatrixMultiply(&Matrix2,
                   &pMeshContainer->pBoneOffsetMatrices[1],
                   pMeshContainer->ppBoneMatrixPtrs[1] );

pDevice->SetTransform ( D3DTS_WORLDMATRIX ( 0 ) , &matrix0 );
pDevice->SetTransform ( D3DTS_WORLDMATRIX ( 1 ) , &matrix1 );
pDevice->SetTransform ( D3DTS_WORLDMATRIX ( 2 ) , &matrix2 );
```

We now see that the values stored in the BoneId array for each subset directly index into our mesh container matrix arrays. This is made possible by the way we store our bone pointers in the same order they are listed in the ID3DXSkinInfo object. We saw how to store the matrices in the mesh container in this order during our software skinning discussion and this process is unchanged.

Of course, for the above code to function properly we will need to enable vertex blending before we render the subset. Since we know that each vertex in our example mesh will have three weights defined and that any unused weights will be set to zero, we can safely set the vertex blend mode as follows and render all subsets of our mesh with this single setting:

```
pDevice->SetRenderState ( D3DRS_VERTEXBLEND , D3DVBF_3WEIGHTS );
```

This will work properly because all of our subset vertices will have three weights defined. This is true even if the subset we are about to render only uses three matrices (two of its three weights) as is the case in the above example where the last element in the BoneId array is set to UINT_MAX. Since the last weight will be set 0.0, any matrix set in the fourth matrix slot on the device will have its contribution scaled to 0.0 and will not effect the position of the transformed vertex. You will notice in the above code example that we only set three of the four possible matrices because the subset only uses three matrices (as described by the BoneId array). The contents of the fourth device matrix are insignificant; even if it is not set to an identity matrix and just contains garbage data, its contribution is reduced to zero by the pipeline so it will have no effect.

The BoneId array allows us to perform a rendering optimization in the non-indexed skinning case. For example, we know that if a subset has a BoneId array of {4, 10, UINT_MAX, UINT_MAX} then we only need to set two matrices, even though the vertex will have three weights. The last two weights will be set to zero, so we do not have to bother setting the third and fourth device matrices as just discussed. This is wasteful however, since every vertex in this subset will still be blended with all four matrices even though the final two matrix multiplies will have no effect on the outcome. Therefore, rather than just setting the D3DRS_VERTEXBLEND renderstate to the number of weights per vertex in the mesh (3 in our example) and rely on the zero weights of the vertex to cancel out the contributions from unused matrix slots, we can check the BoneId array to find out exactly how many matrices will really be needed by this subset. Once we find out this information, we could set the D3DRS_VERTEXBLEND render state to process only this number of vertex weights/matrices.

So in the example subset above, although the vertices contain three weights (because some other subset must have needed to use this many) this particular subset only needs to use two weights for its three matrix contributions. If we set the D3DRS_VERTEXBLEND state to D3DVBF_2WEIGHTS before rendering this subset, we can avoid the overhead of unnecessary per-vertex matrix multiplication. Since

only two of the three defined weights will be used by the pipeline, the fourth matrix slot on the device will be completely ignored.

The following code snippet demonstrates how to test the BoneId array of the current subset we are about to render to see how many matrices will be needed. We then use this value to set the D3DRS_VERTEXBLEND render state so that non-contributing matrices and weights are ignored.

```
      pBoneComb = (LPD3DXBONECOMBINATION)
                        (pMeshContainer->pBoneCombinationBuf->GetBufferPointer());

      // Render each subset
      for (iAttrib = 0; iAttrib < pMeshContainer->NumAttributeGroups; iAttrib++)
      {
            NumBlend = 0;
            for (DWORD i = 0; i < pMeshContainer->NumInfl; ++i)
            {
                  if (pBoneComb[iAttrib].BoneId[i] != UINT_MAX)
                  {
                        NumBlend = i;
                  }
            }

            // Use only relevant matrices / weights
            pDevice->SetRenderState(D3DRS_VERTEXBLEND,  NumBlend);

            // Set matrices and render subset here
       }
```

Now that we know what the bone contribute table in our mesh container will contain, we can continue our discussion of the new members added to our mesh container.

DWORD iAttributeSW

This value will store the zero-based index of the first subset in the mesh that needs to be rendered with software vertex processing enabled. For example, if this value is set to 5, it means that the first five subsets can be rendered with hardware vertex processing and that the remaining subsets (starting from index iAttributeSW up to NumAttributeGroups) need to be rendered with software vertex processing. This is because some of the subsets in that range will need to use more vertex blends than the hardware can support. In software vertex processing mode, we know the pipeline will always have the capability to blend up to four matrices, and the converted mesh will never have any more than three weights defined, so all is fine. However, the hardware may not support all four matrix blends and may instead support only two for example. So if we have a mesh where some of the subsets require more matrix blends than the hardware supports, we will need to render those subsets using software vertex processing.

ConvertToBlendedMesh will try to intelligently arrange the mesh based on the hardware capabilities of the device. For example, if the hardware supports only two weights (three matrix influences) but the original bone/vertex mapping describes one or more vertices as being influenced by more bones than three (four or more), then the function will try to return a mesh that only has three weights defined

without severely compromising the skin/bone relationship. In that case, means we can render the entire mesh using hardware vertex processing.

However, if the device supports a maximum of only two influences per vertex (i.e. one weight) then the function may not be able to approximate the mesh down to this level without severely compromising the relationship between the skin and bones. Therefore some of the subsets in the returned mesh may still require more matrix blends than the device can handle in hardware. It is in this instance that we will use the iAttributeSW flag to distinguish these subsets so that we can optimize the rendering of the mesh using two passes. In the first pass we will render all of the subsets that are influenced by no more than two bones using hardware vertex processing. In the second pass we will enable software vertex processing and render the remaining subsets that have three or more influences. Because we have the index of the first subset that is non-hardware compliant, we do not have to loop through and test all subsets again from the beginning during the second pass. Instead we can start checking subsets from this index.

After we call the ConvertToBlendedMesh function to create the new skinned mesh (in our CreateMeshContainer function), our first job will be to check how many matrix blends the hardware supports. We do this by querying the **MaxVertexBlendMatrices** member of the D3DCAPS9 structure as shown below.

```
D3DCAPS9 d3dcaps;
pDevice->GetDeviceCaps ( &d3dcaps );
DWORD MaxNumberBlends = d3dcaps.MaxVertexBlendMatrices;
```

Once we have this value, we will loop through each subset in the D3DXBONECOMBINATION buffer and search each BoneId array to see if that subset requires more matrix influences than the hardware supports. Once we find the first subset in the buffer that exceeds the hardware capabilities, we will set the iAttributeSW member to this index. Now we can render the mesh using the two pass approach just discussed. In the first pass we will render all subsets in hardware using this logic.

```
for ( int I = 0 ; I < NumSubsets; I++ )
{
     if ( Subset[I] can be rendered in hardware )  RenderSubset[I];
}
```

In this first pass we have looped through every subset and only render subsets that are not influenced by more matrices than the hardware supports. In the next pass, we will do the same again only this time with software vertex processing enabled. However, because we have stored the index of the first non-hardware compliant subset, we do not have to start testing from the beginning. We can start from this index because we know that no subsets exist before this index that cannot be rendered in hardware.

```
pDevice->SetSoftwareVertexProcessing ( TRUE );

for ( int I = iAttributeSoftware ; I < NumSubsets; I++ )
{
      if ( Subset[I] can NOT be rendered in hardware )  RenderSubset[I];
}
```

Fortunately, virtually all recent video cards support more than two matrix blends in hardware, so this will only be an issue on fairly old graphics cards.

11.8.5 ID3DXSkinInfo::ConvertToBlendedMesh

Before we look at our new CreateMeshContainer function for non-indexed skinning, we will examine the ID3DXSkinInfo::ConvertToBlendedMesh function. This function that will take the regular mesh we are passed to our CreateMeshContainer callback function and output a new mesh that has the correct number of weights in the vertices. This function will never return a mesh with more than three weights (four influences) defined for each vertex even if the bone to vertex data stored inside the ID3DXSkinInfo object has vertices that are mapped to more than four bones. In this case, the function will approximate the mesh as best as it can so that it meets this requirement. It will also try to take the current hardware into account so that it will return a mesh that is within the maximum matrix blend capabilities of the device. This was discussed in the previous section.

ConvertToBlendedMesh will be called from inside our CreateMeshContainer function. We will pass in the interface to the standard mesh that we were passed by D3DXLoadMeshHierarchyFromX. This function will use the ID3DXSkinInfo's internal bone and weight information to clone a new mesh into a format that can be vertex blended by the pipeline. The new mesh will still contain all of the old input mesh data such as vertex position, vertex normals, texture coordinates, etc., but now it will have additional information that allows it to use multiple device matrices for blending.

The parameter list might seem pretty overwhelming at first, but most of them are actually output parameters that we can store in our mesh container and use to render the mesh later. We have already covered most of them in the context of the mesh container structure earlier in the lesson.

```
HRESULT ID3DXSKinInfo::ConvertToBlendedMesh
(
    LPD3DXMESH      pMesh,
    DWORD           Options,
    CONST LPDWORD   pAdjacencyIn,
    LPDWORD         pAdjacencyOut,
    DWORD           *pFaceRemap,
    LPD3DXBUFFER    *ppVertexRemap,
    DWORD           *pMaxVertexInfl,
    DWORD           *pNumBoneCombinations,
    LPD3DXBUFFER    *ppBoneCombinationTable,
    LPD3DXMESH      *ppMesh
);
```

LPD3DXMESH pMesh

This first parameter is a pointer to the source mesh interface passed into our CreateMeshContainer method by the D3DX loading function. This mesh is a standard ID3DXMesh that contains the skin geometry and it will be cloned into a blend-enabled mesh by this function. When this function returns and the mesh has been cloned, this mesh will no longer be needed and its interface can be released.

DWORD Options

This parameter is like the options flags used in the other mesh creation and cloning functions we have studied previously. It can be a combination of D3DXMESH flags and D3DXMESHOPT flags. Recall that the D3DXMESH flags allow us to choose properties such as which resource pool the vertex and index buffers of the mesh will be created in and whether or not this mesh should have the capability to be rendered with software vertex processing. The D3DXMESHOPT allows us to specify which optimizations should be applied to the vertex and index data of the cloned mesh. For example, the following flag combination would inform the function to create the vertex buffer and index buffer of the cloned mesh in the managed resource pool with vertex cache optimization. We know from previous lessons that this optimization actually performs the compact optimization, the attribute sort optimization, and will optimize the mesh to achieve better vertex cache coherency where possible.

```
D3DXMESH_MANAGED | D3DXMESHOPT_VERTEXCACHE
```

The optimizations will not be as effective as those applied to a regular mesh because now the triangles will also need to be batched into subsets based on bone contribution. We may have ten triangles which all share the same texture and material and, under normal circumstances, they would all end up in a single subset. But if these triangles do not all use the same N bones, they will be batched into additional subsets based on shared bones.

CONST LPDWORD pAdjacencyIn

This is a DWORD pointer that contains the adjacency information for the source mesh. We do not need to calculate this information ourselves since our CreateMeshContainer function will be passed the adjacency information of the source mesh by D3DXLoadMeshHierarchyFromX.

LPDWORD pAdjacencyOut [output]

Although we can pass NULL as this parameter, if we are interested in the adjacency information of the resulting mesh then we can pass in a pre-allocated DWORD array large enough to hold three DWORDs for every triangle in the source mesh. When the function returns, this array will be populated with the adjacency information of the newly cloned mesh. If you do pass NULL as this parameter, you can always generate the adjacency information for the output mesh at a later time using the ID3DXMesh::GenerateAdjacency method.

DWORD *pFaceRemap [output]

The pFaceRemap parameter should be large enough to hold one DWORD for every face in the source mesh. When the function returns, each element in the array will describe how the index of a face in the source mesh has been mapped to an index in the output mesh. If pFaceMap[5]=10 for example, this means that the 6^{th} face in the original index buffer of the source mesh has now been moved to the 11^{th} face slot in the index buffer of the output mesh. This gives us a chance to update any external structures that we may be maintaining that are linked to the original faces by index. You can set this pointer to NULL if you do not require the re-map information.

LPD3DXBUFFER *ppVertexRemap [output]

The ppVertexRemap parameter points to the address of an ID3DXBuffer interface which on function return will contain an array of DWORDs for each vertex in the source mesh describing how the vertex has been moved in the vertex buffer of the output mesh. The function will create this buffer during the call so we will not pre-allocate it. If you do not need this information then you can set this parameter to NULL.

DWORD *pMaxVertexInfl [output]

For this parameter we pass in the address of a DWORD which on function return will tell us the maximum number of bones that influence a single vertex in the mesh. This is the value that we will store in the NumInfl member of our mesh container. It is significant because subtracting one from this value will tell us how many weights each vertex has defined (we subtract one because the weight for the last influence is always calculated on the fly). If this value returns 4 for example, then it means the vertex structure of this mesh will have three weights defined and there will be at least one subset that needs access to all four matrices on the device. As mentioned previously, this does not mean that all vertices in the mesh need to use four matrices even though they will all store three weights. Any weights in the vertex that are not used will be set to 0.0 so that the corresponding matrix will not contribute.

This value is also significant because it tells us how large each BoneId array is in the D3DXBONECONBINATION structure for each subset in our bone combination table. If this value is 3 for example, then we know that for every element in the bone combination table, each D3DXBONECOMBINATION structure will contain a BoneId array with three elements. Not all elements in the BoneId array for a given subset are necessarily used. If a given subset only needs to use two matrices, its BoneId array would still have 3 elements (in our current example) but the value of the third element would be set to UINT_MAX instead of a valid bone matrix index. This allows us to perform a rendering optimization by only setting the matrices that are used and setting the D3DRS_VERTEXBLEND render state appropriately so that no-op vertex matrix transformations are not carried out.

DWORD *pNumBoneCombinations [output]

There will be one D3DXBONECOMBINATION structure in this buffer for each subset in the output mesh. The bone combination buffer itself will be calculated by this function and returned to us so that we can store it away in our mesh container. This parameter is the address of a DWORD which on function return will tell us how many D3DXBONECOMBINATION structures are stored in the returned ID3DXBuffer (next parameter). Since there will be one D3DXBONECOMBINATION structure for each subset in the output mesh, this value will also tell us how many subsets exist in the output mesh, which we will need to know during rendering.

The value stored in this DWORD will be copied into the NumAttributeGroups member of our mesh container and will be used during rendering to loop through each subset that needs to be rendered.

LPD3DXBUFFER *ppBoneCombinationTable [output]

This parameter is the address of an ID3DXBuffer interface pointer. The function will allocate the ID3DXBuffer and on function return it will store the entire bone combination table for the mesh. For each subset in the output mesh (described by the previous parameter pNumBoneCombinations) there will be a D3DXBONECOMBINATION structure describing exactly which bones need to be set on the device in order to render that subset. These indices are stored inside each structure’s BoneId array and they reference directly into the ppBoneMatrixPtrs and pBoneOffsetMatrices arrays in our mesh container. Each D3DXBONECOMBINATION structure also contains the ranges of vertices and indices in the output mesh vertex and index buffers which belong to a particular subset. The D3DXBONECOMBINATION table also stores an AttributeId which indexes into the D3DXMATERIAL buffer that was passed into CreateMeshContainer.

LPD3DXMESH *ppMesh [output]
The final parameter to this method is the address of an ID3DXMesh interface pointer. On function return, it will point to a mesh that will have pMaxVertexInfl weights defined in each vertex, where each weight will have been assigned its correct value. This is the mesh that we will use for rendering, and as such, the source (input) mesh can be released since it will no longer be needed.

11.8.6 The CreateMeshContainer Function

The first part of the mesh container creation function is completely unchanged from the software skinning version, so we will not need to explore it in much detail here. The following code stores the passed mesh pointer in a newly allocated mesh container (temporary, as this will be replaced with the converted blended mesh) and also copies the name of the mesh passed into the function. Once again, our example will assume that we will need vertex normals in our mesh so that it can be correctly lit by the pipeline. If normals are not present, then we clone the mesh to make space for vertex normals and then calculate them thereafter.

```
HRESULT CAllocateHierarchy::CreateMeshContainer
       ( LPCTSTR Name, LPD3DXMESHDATA pMeshData, LPD3DXMATERIAL pMaterials,
         LPD3DXEFFECTINSTANCE pEffectInstances, DWORD NumMaterials,
         DWORD *pAdjacency,
         LPD3DXSKININFO pSkinInfo, LPD3DXMESHCONTAINER *ppNewMeshContainer )
{
    D3DXMESHCONTAINER_DERIVED *pMeshContainer = NULL;
    UINT iBone, cBones;
    LPDIRECT3DDEVICE9 pd3dDevice = NULL;
    LPD3DXMESH pDestinationMesh = NULL;
    *ppNewMeshContainer = NULL;

    // Get a pointer to the mesh of the skin we have been passed
    pDestinationMesh = pMeshData->pMesh;

    // Allocate a mesh container to hold the passed data.
    // This will be returned from the function
    // where it will be attached to the hierarchy.
    pMeshContainer = new D3DXMESHCONTAINER_DERIVED;
    memset(pMeshContainer, 0, sizeof( D3DXMESHCONTAINER_DERIVED ));

    // If this mesh has a name then copy that into the mesh container too.
    if ( Name ) pMeshContainer->Name = _tcsdup( Name );

    // Get a pointer to the d3d device
    pDestinationMesh->GetDevice(&pd3dDevice);

    // if no normals exist in the mesh, add them ( we might need to use lighting )
    if (!(pDestinationMesh->GetFVF() & D3DFVF_NORMAL))
    {
        pMeshContainer->MeshData.Type = D3DXMESHTYPE_MESH;

        // Clone the mesh to make room for the normals
        pDestinationMesh->CloneMeshFVF( pMesh->GetOptions(),
                                        pMesh->GetFVF() | D3DFVF_NORMAL,
```

```
                                        pd3dDevice,
                                       &pMeshContainer->MeshData.pMesh );

         pDestinationMesh = pMeshContainer->MeshData.pMesh;

        // Now generate the normals for the pmesh
        D3DXComputeNormals( pDestinationMesh, NULL );
    }
   else  // if normals already exist, just add a reference
    {
        pMeshContainer->MeshData.pMesh = pDesinationMesh;
        pMeshContainer->MeshData.Type = D3DXMESHTYPE_MESH;
        pDestinationMesh->AddRef();
    }
```

We now have a regular mesh stored in the mesh container (which is correct because we must remember that the same X file could contain skinned meshes and regular meshes) and at this point we would usually parse the passed texture and material information and store it in some meaningful way. We have already covered how to do this in previous lessons so will not show it here. You can check the accompanying workbook for this lesson for more details.

If we have been passed a valid ID3DXSkinInfo interface, then it means that the passed mesh is to be used to skin the hierarchy (or a section of it) and we will need to convert the passed mesh into a mesh that includes bone weights.

The first section of the code that is executed when we have been passed a skinned mesh is also unchanged -- we store the ID3DXSkinInfo pointer in our mesh container for later use since we will need it after the hierarchy is loaded to map bone offset matrices to absolute bone matrices. We then increase the reference count because we have made a copy of the interface, and we retrieve the bone offset matrices and store them in the mesh container. All of this is unchanged from the software skinning case.

```
    // If there is skinning information,
    // save off the required data and then set up for skinning
    if (pSkinInfo != NULL)
    {
        // first save off the SkinInfo and original mesh data
        pMeshContainer->pSkinInfo = pSkinInfo;
        pSkinInfo->AddRef();

        // We now know how many bones this mesh has
        // so we will allocate the mesh containers
        // bone offset matrix array and populate it.
        NumBones = pSkinInfo->GetNumBones();
        pMeshContainer->pBoneOffsetMatrices = new D3DXMATRIX  [NumBones];

        // Get each of the bone offset matrices so that
        // we don't need to get them later
        for (iBone = 0; iBone < NumBones; iBone++)
        {
             pMeshContainer->pBoneOffsetMatrices[iBone] =
           *(pMeshContainer->pSkinInfo->GetBoneOffsetMatrix(iBone));
        }
```

The next section is where things change compared to software skinning. It is time to convert the mesh that we were passed into a blend-enabled mesh using ID3DXSkinInfo::ConvertToBlendedMesh. We currently have the source mesh stored in the MeshData member of the mesh container. This is where we want the resulting mesh to be stored, so we copy the source mesh pointer to a temporary SourceMesh pointer.

```
    ID3DXMesh * pSourceMesh = pMeshContainer->MeshData.pMesh;
    pMeshContainer->MeshData.pMesh=NULL;

    pMeshContainer->pSkinInfo->ConvertToBlendedMesh
                                (
                                    pSourceMesh,
                                    D3DXMESH_MANAGED|D3DXMESHOPT_VERTEXCACHE,
                                    pAdjacency,
                                    NULL, NULL, NULL,
                                    &pMeshContainer->NumInfl,
                                    &pMeshContainer->NumAttributeGroups,
                                    &pMeshContainer->pBoneCombinationBuf,
                                    &pMeshContainer->MeshData.pMesh
                                );

    pSourceMesh->Release ();
```

We pass in the source mesh and request that the resulting mesh be created as a managed resource which has the vertex cache optimization applied to it. We also pass in the adjacency information that was passed to CreateMeshContainer function and pass NULL for the following three parameters (we will have no need for the adjacency, vertex re-map, or face re-map information that this function generates for the output mesh). In the mesh container's NumInfl member will be stored the maximum number of bones that influence a single vertex in the mesh (which also tells us indirectly how many weights are stored in each vertex) and in the NumAttributeGroups member we store the number of subsets in the output mesh. The bone combination buffer is stored in the mesh container's pBoneCombinationBuf member and the converted output mesh is stored in the mesh container's MeshData.pMesh member.

Finally, we release the source mesh; we will be using the new mesh to render from this point forward because it contains the bone weight information the vertex blending module of the transformation pipeline needs for proper transformations.

At this point we have a mesh that contains the correct number of weights, but it is possible that some subsets in the output mesh may require the use of more matrices than is supported by the hardware. Recall that when the device is in software vertex processing mode we will always be able to render using up to four matrix influences. So taking this into account, before we return from the CreateMeshContainer function, we will loop through each subset in the output mesh and examine each subset's bone combination table. For each subset, we will examine the BoneId array of its D3DXBONECOMBINATION structure to count how many bone influences affect it. As soon as we find a subset that has more bone influences than the hardware supports, we will store the index of this subset in the mesh container's iAttributeSoftware member. All subsets up to this index can be rendered using hardware vertex processing and all subsets after that index (itself included) *may* have to be rendered with software vertex processing.

We start by getting a pointer to the mesh container's bone combination table which is stored inside an ID3DXBuffer:

```
LPD3DXBONECOMBINATION rgBoneCombinations  =
  (LPD3DXBONECOMBINATION)(pMeshContainer->pBoneCombinationBuf->GetBufferPointer());
```

We will now loop through each subset in the mesh:

```
 for (pMeshContainer->iAttributeSW = 0;
      pMeshContainer->iAttributeSW < pMeshContainer->NumAttributeGroups;
      pMeshContainer->iAttributeSW++ )
     {
```

For each subset we will loop through each element of its BoneId array which contains bone matrix indices that are used by this subset. If for example, the NumInfl member returned by the ConvertToBlendedMesh equals 4, then it means that the BoneId array will be 4 elements long (i.e., large enough to hold four matrix indices). If a particular subset only uses two bone matrices, the third and fourth elements will be set to UINT_MAX to indicate that they are unused. We can use this fact to count how many matrix blends the pipeline will need to perform to render this subset, as shown below.

```
            DWORD cInfl   = 0;
            for (DWORD iInfl = 0; iInfl < pMeshContainer->NumInfl; iInfl++)
            {
                if (rgBoneCombinations[pMeshContainer->iAttributeSW].BoneId[iInfl]
                   != UINT_MAX)
                {
                    ++cInfl;
                }
            }
```

At this point in the code, the DWORD local variable cInfl will tell us how many matrix blends the current subset we are checking will require. If this is larger than the maximum number of matrix blends supported by the hardware then we will break from the loop since we have just found the first subset that the hardware will not support. The index of this subset will be stored in the mesh container's iAttributeSoftware member because this was used as the loop variable to iterate through each subset.

```
            if (cInfl > m_d3dCaps.MaxVertexBlendMatrices)
            {
                break;
            }
        }
```

The iAttributeSoftware variable will now contain either the index of the first subset that cannot be rendered using hardware vertex processing or it will be equal to the number of subsets in the mesh if a non-hardware compliant subset was not found and the loop continued to completion. So if iAttributeSoftware is not equal to the number of subsets in the mesh then it means that part of this mesh will need to be rendered using software vertex processing.

Note however that the current mesh cannot be rendered using software vertex processing because when we create a mesh which we intend to use with software vertex processing on a mixed-mode device, we

must use the D3DXMESH_SOFTWAREPROCESSING mesh creation flag (which we did not do previously). Therefore, if this mesh needs to be software vertex processed, we will need to clone the mesh using this flag so that the resulting mesh can be properly rendered.

```
if (pMeshContainer->iAttributeSW < pMeshContainer->NumAttributeGroups)
{
   LPD3DXMESH pMeshTmp;
   pMeshContainer->MeshData.pMesh->CloneMeshFVF
   (
       D3DXMESH_SOFTWAREPROCESSING|pMeshContainer->MeshData.pMesh->GetOptions(),
       pMeshContainer->MeshData.pMesh->GetFVF(),
       m_pd3dDevice,
       &pMeshTmp
   );

   pMeshContainer->MeshData.pMesh->Release();
   pMeshContainer->MeshData.pMesh = pMeshTmp;
   pMeshTmp = NULL;
  }
 } // end if ID3DXSkinInfo!=NULL
} // End Function
```

The code clones the mesh into a temporary mesh and then releases the old mesh before assigning the newly cloned software processing enabled mesh to the mesh container's MeshData.pMesh member.

That brings us to the end of our non-indexed version of the CreateMeshContainer function. When D3DXLoadMeshHierarchyFromX returns control back to our application, the hierarchy will be loaded and all of our skinned meshes will be stored in their respective mesh containers along with their bone combination tables. Notice how the CreateMeshContainer function no longer requires the code that was used in the software skinning case that allocated (and resized) the temporary global matrix array. That array was only used to pass matrices into the UpdateSkinnedMesh function, which is not used when performing hardware skinning.

Do not forget that after D3DXLoadMeshHierarchyFromX returns control back to your program, you will still need to traverse the hierarchy and store the absolute bone matrices of each frame in the mesh container's ppBoneMatrixPtrs array, just as we did in the software skinning technique. You will recall that we wrote a function to do this called SetupBoneMatrixPointers and this function is unchanged regardless of the skinning technique used.

We have now seen everything there is to see with regards to setting up the hierarchy and storing the skinned meshes in a format which can be used by the non-indexed vertex blending module of the pipeline.

11.8.7 Transforming and Rendering the Skin

Animating the hierarchy is no different from the software skinning case (or in fact any animation of the frame hierarchy) -- we use the animation controller's AdvanceTime method to apply the animation data to the relative bone matrices of the hierarchy and then perform a pass through the hierarchy to recalculate the absolute matrix for each frame. So the next and final stage of the non-indexed skinning technique we need to look at is the code that would be used to actually render the mesh. We already know how to recursively traverse the hierarchy and call the DrawMeshContainer function for each mesh container found, so we will focus on this DrawMeshContainer function, just as we did in the software skinning technique.

Our drawing function in this example is passed a pointer to the mesh container to be rendered and a pointer to the owner frame. This frame is important if this is not a skinned mesh because this frame's combined matrix will be the matrix we wish to use as the world matrix to render the mesh.

The first thing this function does is cast the passed mesh container and frame to our derived class types.

```
void DrawMeshContainer (LPD3DXMESHCONTAINER pMeshContainerBase,
                        LPD3DXFRAME pFrameBase)
{

    D3DXMESHCONTAINER_DERIVED*pMeshContainer =
                                  (D3DXMESHCONTAINER_DERIVED*)pMeshContainerBase;

    D3DXFRAME_MATRIX *pFrame = (D3DXFRAME_MATRIX*)pFrameBase;

    UINT NumBlend;
    UINT iAttrib;
    LPD3DXBONECOMBINATION pBoneComb;
    UINT iMatrixIndex;
    D3DXMATRIXA16 matTemp;
    IDirect3DDevice9 * pDevice;

    pMeshContainer->MeshData.pMesh->GetDevice(&pDevice);
```

This function can be called for both skinned meshes and regular meshes so we will need to make sure (as we did in the software skinning version of this function) that we cater to both types. Because we are covering each of the skinning techniques in isolation for the purpose of clarity, in this version of the code we are assuming that if the ID3DXSkinInfo pointer stored in the mesh container is not NULL, then the mesh stored here is setup to be used with the non-indexed skinning technique.

The first thing we do is fetch the bone combination table of the mesh container so that we can get the matrix indices needed to render each subset.

```
    // first check if this is a skinned mesh or just a regular mesh
    if (pMeshContainer->pSkinInfo != NULL)
    {
```

```
        pBoneComb = ( LPD3DXBONECOMBINATION )
                    ( pMeshContainer->pBoneCombinationBuf->GetBufferPointer() );
```

Our next task will be very similar to the code we used to calulate the index of the iAttributeSoftware flag. For each subset we will first count how many matrices it requires for rendering. We will consider this the first pass over the mesh where we draw all subsets that can be rendered using hardware vertex processing.

```
        // Draw all subsets that can be rendered with Hardware Processing First
        for (iAttrib = 0; iAttrib < pMeshContainer->NumAttributeGroups; iAttrib++)
        {
               NumBlend = 0;
               for (DWORD i = 0; i < pMeshContainer->NumInfl; ++i)
               {
                   if (pBoneComb[iAttrib].BoneId[i] != UINT_MAX)
                   {
                       NumBlend = i;
                   }
                }
```

For each subset we will check to see whether or not the number of matrices that influence the subset (stored in the NumBlend local variable) is within the capabilities of the hardware. If so, we will fetch the matrix indices from this subset's BoneId array and use them to fetch the corresponding bone matrix and bone offset matrix stored in the mesh container. We will combine these two matrices to create a single world matrix and set that matrix in the relevant position in the device matrix palette (indices 0 through 3). The position of the index in the BoneId array describes which of the four matrix slots on the device the matrix should be bound to, as shown below.

```
               // Can we render this subset in hardware?
               if (m_d3dCaps.MaxVertexBlendMatrices >= NumBlend + 1)
               {
                   // Fetch each bone matrix for this subset.
                   // Combine with its offset matrix, and set it
                   for (DWORD i = 0; i < pMeshContainer->NumInfl; ++i)
                   {
                       iMatrixIndex = pBoneComb[iAttrib].BoneId[i];
                       if (iMatrixIndex != UINT_MAX)
                       {
                           D3DXMatrixMultiply( &matTemp,
                              &pMeshContainer->pBoneOffsetMatrices[iMatrixIndex],
                               pMeshContainer->ppBoneMatrixPtrs[iMatrixIndex] );

                           pDevice->SetTransform( D3DTS_WORLDMATRIX( i ),
                                                  &matTemp );
                       }
                  } // End for each matrix influence
```

At this point, the matrices are correctly set on the device for the current subset we are about to render and the local NumBlend variable tells us how many matrices have been set for this subset. This allows us to optimize the rendering by setting the D3DRS_VERTEXBLEND to process only the required number of matrices, instead of wastefully processing every weight defined in the vertex. We finally render the current subset (and repeat for every subset in the mesh).

```
                    pDevice->SetRenderState(D3DRS_VERTEXBLEND, NumBlend );

                    // lookup the material used for this subset of faces
                    Set Textures and Materials Here ( Not Shown )

                    pMeshContainer->MeshData.pMesh->DrawSubset(iAttrib);

                } // End can be rendered in hardware

            } // End for each attribute
```

At this point we will have looped through each subset and will have rendered all subsets that can be rendered using hardware vertex processing. However, some subsets may not have been rendered in the above loop if they required more matrix blends than the hardware supported. Therefore, we will now loop through the subsets and basically do what the above code did all over again. This time however, we will enable software vertex processing and render only the subsets that have more bone influences than the hardware supports. This is the second pass we discussed earlier in the lesson. Because we have stored (in the mesh container's iAttributeSoftware member) the index of the first subset that exceeds the hardware capabilities, we do not have to loop through every subset this time -- we just have to start from this index.

```
    // If necessary draw any subsets that the hardware could not handle
    if (pMeshContainer->iAttributeSW < pMeshContainer->NumAttributeGroups)
    {
       pDevice->SetSoftwareVertexProcessing(TRUE);

         for ( iAttrib = pMeshContainer->iAttributeSW;
               iAttrib < pMeshContainer->NumAttributeGroups;
               iAttrib++)
         {
               NumBlend = 0;
               for (DWORD i = 0; i < pMeshContainer->NumInfl; ++i)
               {
                   if (pBoneComb[iAttrib].BoneId[i] != UINT_MAX)
                   {
                        NumBlend = i;
                   }
               }

               if (m_d3dCaps.MaxVertexBlendMatrices < NumBlend + 1)
               {
                   // Fetch each bone matrix for this matrix.
                   // Combine with its offset matrix. And set it
                   for (DWORD i = 0; i < pMeshContainer->NumInfl; ++i)
                   {
                       iMatrixIndex = pBoneComb[iAttrib].BoneId[i];
                       if (iMatrixIndex != UINT_MAX)
                       {
                           D3DXMatrixMultiply( &matTemp,
                                &pMeshContainer->pBoneOffsetMatrices[iMatrixIndex],
                                 pMeshContainer->ppBoneMatrixPtrs[iMatrixIndex] );

                           pDevice->SetTransform( D3DTS_WORLDMATRIX( i ),
```

```
                                                &matTemp );
                        }
                    }

                    m_pd3dDevice->SetRenderState(D3DRS_VERTEXBLEND, NumBlend );

                    // Set Textures and Materials for the current Subset  Here

                    // draw the subset -- correct material/matrices are loaded
                    pMeshContainer->MeshData.pMesh->DrawSubset(iAttrib);

                } // end if can be rendered in hardware

            }  // end for each subset

            pDevice->SetSoftwareVertexProcessing(FALSE);

    } // end if software vertex processing subsets exist in this mesh
```

Finally, we disable vertex blending just in case any other objects in our scene need to be rendered.

```
                pDevice->SetRenderState(D3DRS_VERTEXBLEND, 0);
    } // end if this is a skinned mesh
```

If this was not a skinned mesh, then we render the mesh normally as shown below.

```
    else  // this is just a regular mesh so draw normally
    {
            pDevice->SetTransform(D3DTS_WORLD, &pFrame->mtxCombined);
            DrawMesh ( pMeshContainer->MeshData.pMesh )
    }
            pDevice->Release();

} // end function
```

And that is all there is to hardware non-indexed skinning. Although the code is fairly long, this is only because we repeat a lot of it to handle the two separate passes.

11.8.8 Non-Indexed Skinning Summary

Non-indexed Skinning: Advantages

- Vertex blending is performed in hardware by the GPU, making it faster than software skinning.
- We have the ability to optimize rendering of a subset by setting the D3DRS_VERTEXBLEND renderstate so that only the influential matrices are used. This

avoids the need for the pipeline to factor in matrix calculations that have zero weights which would not contribute towards the final world space position of the vertex.

- This form of hardware skinning is very well supported even on older T&L graphics hardware, although some hardware may not support all four possible matrix blends. Even if you do decide to support indexed skinning (discussed next) you should always implement this skinning technique as a fallback option because indexed skinning is not as widely supported in hardware.
- This skinning method can be used even if hardware support is not available as long as we create either a software or mixed-mode device. In this case, the vertex blending will be done in the software vertex blending component of the pipeline. This means that we can implement just this technique and have it work even when no hardware support is available for vertex blending.

Non-indexed Skinning: Disadvantages

- Only four bone influences per vertex (although this is usually more than enough).
- Because there is no information within each vertex specifying how each weight in the vertex maps to matrices in the device matrix palette, an implicit mapping is used where each of the four weights map to the first four matrices in the palette. This means that we can only use four out of the 256 matrix slots on the device. This results in the mesh being broken into many more subsets because each subset can only contain triangles that are influenced by the same four bone matrices.
- Only four bone influences per subset.
- Batch rendering severely compromised.

11.9 DirectX Skinning III: Indexed Skinning

The final skinning technique we will cover in this chapter is indexed skinning. Just like the non-indexed skinning technique, the geometry blending mechanism used is integral to the DirectX transformation pipeline. Indexed skinning is the most desirable skinning technique because it inherits the best aspects from both software skinning and pipeline skinning without the limitations of the non-indexed approach. Unfortunately, indexed skinning is not as widely supported in older hardware, but because it is part of the DirectX transformation pipeline, even if hardware support is not available, we can still perform this skinning technique with software vertex processing (assuming we have created a device for which software vertex processing is applicable).

On the bright side, indexed skinning works almost identically to its non-indexed cousin as far as implementation details go. So we will only have to cover a few changes to add indexed skinning support to our current code base.

In the previous section we learned that one of the biggest disadvantages of using non-indexed skinning was that we can only use four out of the 256 possible device palette world matrix slots. This was because each vertex in the blend-enabled mesh contains a number of weights, but no information describing which matrix slots on the device those weights are applicable to. The result was that a one to one mapping is implied, such that weights 0 – 3 in the vertex are always used to weight the contributions of world matrices 0 – 3 on the device (actual indices = 256 – 259). So even though the device supports 256 world matrices, any subset that we render could only consist of vertices that used the first four. One undesirable side effect was that batching was seriously compromised. Even if the entire mesh used only a single texture and material (which would usually cause all faces to be grouped into a single subset), the mesh would still have to be broken into multiple subsets, where each subset contained only the triangles that share the same four bone matrices.

When we think about the non-indexed case for a moment and lament the wasted matrices in the device matrix palette, we soon realize that our problems could all be solved if we had the ability to store matrix indices in each vertex. For example, if a vertex has N weights, then we would need to store N+1 matrix indices (because we recall that the N+1 weight is calculated on the fly in the pipeline). If we could do this, then when the pipeline transforms a vertex, it can examine each weight and its corresponding matrix index (the index would be a value between 0 and 255 describing which matrix in the device matrix palette the weight is defined for) and use the correct matrix to transform the vertex. If we had a mesh with 50 bones where all faces used the same texture, we could have a single subset because we would then have the ability to set all 50 bone matrices in the matrix palette simultaneously before rendering the mesh. Thus our batching optimizations are left intact. And if, practically speaking, each subset was no longer limited by how many matrices it can use and we consider that a vertex can still have four bone influences, then any given triangle could actually be manipulated by up to 12 unique bone matrices.

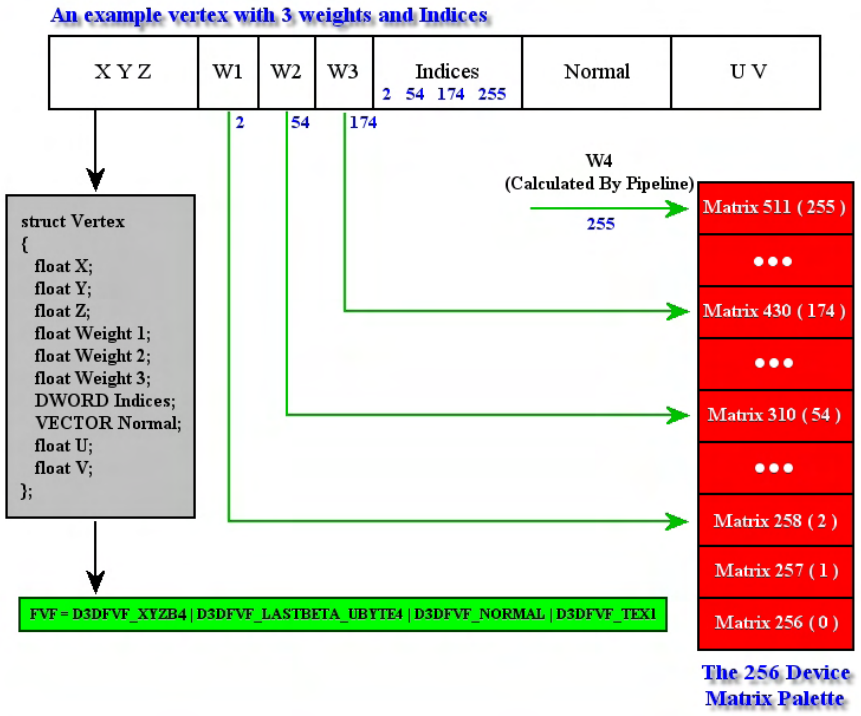


Figure 11.21

It just so happens that there is an FVF flag that allows us to place up to four matrix indices in our vertex structure in addition to the vertex weights. In Fig 11.21, the vertex has three weights (i.e. four bone influences) along with a new DWORD member called Indices. In each of the four bytes of the DWORD the index of one of the weights is stored. This tells the pipeline which matrix in the palette should be used with a given weight in the vertex to transform the mesh. In the example above, you can see that the vertex is to be transformed by the matrices stored in world matrix slots {2, 54, 174, 255}. These are the matrices used to transform this vertex for weights 1 through 4 respectively. We can imagine how another vertex in the same triangle might have a completely different set of indices, and this would hold true for every other vertex within the subset. Before rendering a subset, we just need to set all of the matrices that will be used by it (possibly even the entire bone hierarchy) in their respective matrix slots and then draw it.

11.9.1 Storing Matrix Palette Indices

Before continuing, we will take a look at the FVF flags that must be used to describe a vertex structure that includes both bone weights and matrix palette indices. These indices will take the form of the four individual bytes of a single DWORD. You would normally need to know these FVF flags when creating the vertex buffer that will store the indexed skin, but as it happens we can use a method of the ID3DXSkinInfo interface to work this out on our behalf. Like the ConvertToBlendedMesh function that we studied previously, which clones a normal mesh into one that includes weights, there is a second method that we will use for indexed skinning called ConvertToIndexedBlendedMesh. It does virtually the same thing as its sibling, only this time returns a mesh that has weights and indices defined in each vertex. Therefore, setting up the mesh for the indexed skinning technique is essentially identical to the non-indexed case. The exception is that we will call ConvertToIndexedBlendedMesh in our CreateMeshContainer function instead of the non-indexed version. Nevertheless, we will still look at the correct FVF flags that would be needed to create a vertex buffer that supports both weights and matrix palette indices just in case you need to build the buffer by hand.

Just for review, a vertex structure that would allow for three weights (four bone influences) is shown below along with the accompanying FVF flags.

```
struct NonIndexVertex
{
   D3DXVECTOR3 Pos;
   float Weight1;
   float Weight2;
   float Weight3;
   D3DXVECTOR3 Normal;
};

FVF = D3DFVF_XYZB3 | D3DFVF_NORMAL;
```

Recall that D3DFVF_XYZB3 informs the pipeline that the vertex will have an untransformed XYZ position and three float weights defined. Following the weights in this example we have also defined a vertex normal.

If we want to use the same basic structure, but include a matrix index for each weight then we have:

```
struct IndexVertex
{
   D3DXVECTOR3 Pos;
   float Weight1;
   float Weight2;
   float Weight3;
   DWORD Indices;
   D3DXVECTOR3 Normal;
};
```

The FVF for this structure is slightly less intuitive. There is a new FVF flag being used along with a change to one of the others.

```
FVF = D3DFVF_XYZB4 | D3DFVF_LASTBETA_UBYTE4 | D3DFVF_NORMAL;
```

The first thing you will notice is that we are now describing the vertex structure as containing four weights instead of three (D3DFVF_XYZB4). Technically this is not true of course since we only used three weights, but there is a catch. Since a DWORD and a FLOAT are both four bytes each in size, the above vertex data structure could be rewritten as:

```
struct IndexVertex
{
   D3DXVECTOR3 Pos;
   float Weight1;
   float Weight2;
   float Weight3;
   float Indices;
   D3DXVECTOR3 Normal;
};
```

Or we could even write the structure like so:

```
struct IndexVertex
{
   D3DXVECTOR3 Pos;
   float Weights[4];
   D3DXVECTOR3 Normal;
};
```

So we can see that while we might *define* the indices member as a DWORD (in the first structure) following the three weights to increase programmer readability, as far as the pipeline is concerned we are just passing it another four bytes. Therefore, our structure now has *space* for four weights defined.

Note that since a vertex cannot be influenced by more than four matrices, defining four weights would actually break this rule because it says it is influenced by five matrices. So our example flag would not actually work at all in the non-indexed case. The secret to the whole process is in the use of the second FVF flag shown above: D3DFVF_LASTBETA_UBYTE4. This flag tells the pipeline that the last weight (a weight is also sometimes called a *beta*) will be treated differently. It informs the pipeline that if it has been instructed to blend using N blend weights in the vertex, the Nth blend weight should not be used as a blend weight, but instead should be used as a four byte matrix palette index container.

So our FVF set above works with the vertex type we saw in Fig 11.21 -- a vertex with three actual blend weights (four bone contributions) and an additional four byte area containing the indices of up to four matrices in the device palette.

However, it must be made absolutely clear that D3DFVF_LASTBETA_UBYTE4 does not say that the last weight *defined in the vertex* will be the weight that will be used as an index container. Instead it says that the last weight that the pipeline has been instructed to process during vertex blending will be used as the index container. In other words, the last weight (last beta) is determined by the D3DRS_VERTEXBLEND render state and not by the number of weights defined in the vertex format. For example, if the D3DRS_VERTEXBLEND renderstate is set to D3DVBF_2WEIGHTS so that the pipeline will only process two weights for the vertex (three bone influences), the last beta in this example would be the third weight defined in the vertex (weight[2]). This weight would be treated as a DWORD with 4-byte indices. This is true even if the vertex contained four weights. So it would not be correct in our example, where the index data would be stored in the fourth weight, not the third.

We can see then that the D3DRS_VERTEXBLEND render state will also have to be set appropriately before rendering the mesh so that the correct weights are used for blending and the correct weight is used to feed matrix index data to the pipeline. Thus, we can no longer use the optimization that we did in the non-indexed case where we toggled the D3DRS_VERTEXBLEND render state between subsets to rid ourselves of zero weight matrix contributions. If we store our indices in the fourth weight for example, we will need this to be interpreted as the Indices member for all vertices that we render as defined by our vertex structure, regardless of the fact that there may be no-ops that result. The loss of this optimization is more than made up for however by the massive gains made from having much larger subsets.

Here is another example of a structure that will be indexed blended using three bone contributions (i.e. two weights).

```
struct IndexVertex
{
   D3DXVECTOR3 Pos;
   float Weight1;
   float Weight2;
   DWORD Indices;
   D3DXVECTOR3 Normal;
};

FVF = D3DFVF_XYZB3 | D3DFVF_LASTBETA_UBYTE4 | D3DFVF_NORMAL;
```

Our final example is an indexed vertex that has two bone contributions (1 weight) and its corresponding flags.

```
struct IndexVertex
{
   D3DXVECTOR3 Pos;
   float Weight1;
   DWORD Indices;
   D3DXVECTOR3 Normal;
};

FVF = D3DFVF_XYZB2 | D3DFVF_LASTBETA_UBYTE4 | D3DFVF_NORMAL;
```

11.9.2 Determining Support for Indexed Skinning

Before creating our mesh to use indexed skinning, we will need to determine what level of support the hardware provides. If the device is in software vertex processing mode, then indexed skinning will always be supported and we will be able to access all 256 world matrices on the device. But when using hardware vertex processing, things are not so simple.

First, if the device supports indexed blending at all, it may only support a subset of the 256 possible matrices. For example, a hardware device might support indexed vertex blending but only use the first 64 world matrix device slots. This is information we will need to know because it affects the way ConvertToIndexedBlendedMesh calculates the skinned mesh. If only 64 matrix palette entries are available, then the mesh will need to be adjusted to work with this limitation. This will typically involve breaking the mesh into separate subsets so that no single subset requires more than 64 matrices to be set simultaneously in order to render it. Therefore, when we call the ConvertToIndexedBlendedMesh function from our CreateMeshContainer function to convert our mesh into an indexed skinning compliant mesh, we will also pass in the number of matrices that the current device can support so that the function can make the appropriate adjustments.

We can query the device indexed skinning support by checking the MaxVertexBlendMatrixIndex flag in the D3DCAPS9 structure.

```
D3DDCAPS9 caps;
pDevice->GetCaps ( &caps );
DWORD MaxMatIndex = caps.MaxVertexBlendMatrixIndex;
if ( MaxMatIndex == 0)
{
    //No Hardware Support Available, Enable Software Processing
}
```

The MaxVertexBlendMatrixIndex member contains the value of the maximum index into the matrix palette that the device supports. For example, if this value is set to 63 then it means that this device does support hardware skinning but the pipeline can only use world matrices 0 through 63. Therefore, a single subset in the mesh can be influenced by no more than 64 matrices. In this example, the ConvertToIndexedBlendedMesh function would create a skinned mesh in which every subset uses no more than the first 64 matrices. This might result in more subsets than would otherwise be needed.

If MaxVertexBlendMatrixIndex is set to 0, then indexed geometry blending is not supported by the hardware at all. In this case we can choose to either fallback to software vertex processing or use the non-indexed skinning technique discussed earlier.

11.9.3 Storing the Skinned Mesh

Let us now look at how we might declare our mesh container structure to store the information needed to perform indexed skinning. Note that the frame hierarchy is stored in exactly the same way as we saw with the previous skinning techniques; it is only the way we create and render the mesh that has changed.

```
struct D3DXMESHCONTAINER_DERIVED: public D3DXMESHCONTAINER
{

    DWORD               NumAttributeGroups;
    DWORD               NumInfl;
    LPD3DXBUFFER        pBoneCombinationBuf;
    D3DXMATRIX**        ppBoneMatrixPtrs;
    D3DXMATRIX*         pBoneOffsetMatrices;
    DWORD               NumPaletteEntries;
    bool                UseSoftwareVP;
 };
```

There are two new members in our mesh container; the rest of the members have the same meaning they had in the non-indexed case.

DWORD NumPaletteEntries

This DWORD will store the size of the device matrix palette that is available for skinning. We will calculate this ourselves by querying the D3DCAPS9::MaxVertexBlendMatrixIndex. We will also have to factor in whether or not the mesh has vertex normals. As it happens, if the mesh does have vertex normals, then the number of matrices that the device can use is cut in half. If a device supports 200 matrices (MaxVertexBlendMatrixIndex = 199) then we can only use 100 of these matrices when the mesh has vertex normals because the normals will also have to be transformed. Therefore, if our mesh has normals, we will divide the MaxVertexBlendMatrixIndex value by two before passing it into ConvertToIndexedBlendedMesh. The ConvertToIndexedBlendedMesh function needs to know how many matrices are available for indexed skinning (i.e. the palette size) so that it can force the output mesh to use no more than this many matrices per subset. As discussed previously, if a subset needs to use more matrices than the device currently supports, the mesh will be subdivided into smaller subsets where each subset only uses the number of matrices available.

bool UseSoftwareVP

We will use this simple Boolean to indicate whether the device supports indexed skinning. If not, then we will have to place the device into software vertex processing mode in order to render it using the indexed skinning technique. When software vertex processing is used, the device will always support 256 world matrices (128 if the mesh has vertex normals).

11.9.4 The CreateMeshContainer Function

The first section of our updated CreateMeshContainer function is completely unchanged. We allocate the mesh container, copy the name of the mesh and, in this example, clone the source mesh if it does not contain vertex normals. This is not a requirement for skinning, but since we intend to use the DirectX lighting pipeline in our code, we will get this step out of the way now.

```
HRESULT CAllocateHierarchy::CreateMeshContainer
       ( LPCTSTR Name, LPD3DXMESHDATA pMeshData, LPD3DXMATERIAL pMaterials,
         LPD3DXEFFECTINSTANCE pEffectInstances, DWORD NumMaterials,
         DWORD *pAdjacency,
         LPD3DXSKININFO pSkinInfo, LPD3DXMESHCONTAINER *ppNewMeshContainer )
{
    D3DXMESHCONTAINER_DERIVED *pMeshContainer = NULL;
    UINT iBone, cBones;
    LPDIRECT3DDEVICE9 pd3dDevice = NULL;
    LPD3DXMESH pDestinationMesh = NULL;
    *ppNewMeshContainer = NULL;

    // Get a pointer to the mesh of the skin we have been passed
    pDestinationMesh = pMeshData->pMesh;

    // Allocate a mesh container to hold the passed data.
    // This will be returned from the function
    // where it will be attached to the hierarchy.
    pMeshContainer = new D3DXMESHCONTAINER_DERIVED;
    memset(pMeshContainer, 0, sizeof( D3DXMESHCONTAINER_DERIVED ));

    // If this mesh has a name then copy that into the mesh container too.
    if ( Name ) pMeshContainer->Name = _tcsdup( Name );

    // Get a pointer to the d3d device.
    pDestinationMesh->GetDevice(&pd3dDevice);

    // if no normals exist in the mesh, add them ( we might need to use lighting )
    if (!(pDestinationMesh->GetFVF() & D3DFVF_NORMAL))
    {
        pMeshContainer->MeshData.Type = D3DXMESHTYPE_MESH;

        // Clone the mesh to make room for the normals
        pDestinationMesh->CloneMeshFVF( pMesh->GetOptions(),
                                        pMesh->GetFVF() | D3DFVF_NORMAL,
                                        pd3dDevice,
                                       &pMeshContainer->MeshData.pMesh );

         pDestinationMesh = pMeshContainer->MeshData.pMesh;

        // Now generate the normals for the pmesh
        D3DXComputeNormals( pDestinationMesh, NULL );
    }
    else  // if normals already exist, just add a reference
    {
        pMeshContainer->MeshData.pMesh = pDesinationMesh;
```

```
        pMeshContainer->MeshData.Type = D3DXMESHTYPE_MESH;
        pDestinationMesh->AddRef();
    }
```

We now have a regular mesh stored in the mesh container, so it is time to check to see if we have been passed skinning information. When this is the case, we will start off with the same approach as in the previous version of the function -- we will store the ID3DXSkinInfo interface and bone offset matrices in our mesh container.

```
    // If there is skinning information, save off the required data
    // and then set up for skinning
    if (pSkinInfo != NULL)
    {
        // first save off the SkinInfo and original mesh data
        pMeshContainer->pSkinInfo = pSkinInfo;
        pSkinInfo->AddRef();

        // We now know how many bones this mesh has
        // so we will allocate the mesh containers
        // bone offset matrix array and populate it.
        NumBones = pSkinInfo->GetNumBones();
        pMeshContainer->pBoneOffsetMatrices = new D3DXMATRIX  [NumBones];

        // Get each of the bone offset matrices so that
        // we don't need to get them later
        for (iBone = 0; iBone < NumBones; iBone++)
        {
             pMeshContainer->pBoneOffsetMatrices[iBone] =
                 *(pMeshContainer->pSkinInfo->GetBoneOffsetMatrix(iBone));
        }
```

Now things will start to get a little different. We need to know if the current hardware can perform indexed skinning as well as any matrix limitations that may exist. If the device does not support enough matrices for this mesh, then software vertex processing will have to be invoked later to render it. Imagine a device that supports only six matrices to use for indexing. At first, this might not seem like a problem because we know that a single vertex can only be influenced by four matrices at most. However, our DrawPrimitive rendering units in this case are triangles, not single vertices. So the smallest item we can batch contains three vertices. If each of these three vertices used different matrices, then that triangle would need access to 12 matrices in order to be transformed. This would be beyond the capabilities of our example device so we would have to use software vertex processing.

How do we know if a single triangle in this mesh uses more matrices than the hardware has to offer? It just so happens that ID3DXSkinInfo has a method that calculates the maximum number of bones that influence a single triangle in the mesh. This function is called GetMaxFaceInfluences and is shown next.

```
HRESULT GetMaxFaceInfluences
(
    LPDIRECT3DINDEXBUFFER9 pIB,
    DWORD NumFaces,
    DWORD *maxFaceInfluences
);
```

Because ID3DXSkinInfo only contains the bone-to-vertex information, it has no idea how those vertices are arranged into faces in the mesh. So we must pass in the index buffer followed by the number of triangles in the buffer so that this can all be worked out internally. The third parameter is the address of a DWORD which on function return will contain the maximum number of bones that influence a single triangle in the mesh. We can use this value to determine whether the current hardware can cope with this mesh.

The code that calls this function to fetch the maximum face influence of the mesh follows:

```
        DWORD NumMaxFaceInfl;
        DWORD Flags = D3DXMESHOPT_VERTEXCACHE;
        LPDIRECT3DINDEXBUFFER9 pIB;
        pMeshContainer->MeshData.pMesh->GetIndexBuffer(&pIB);
        pMeshContainer->pSkinInfo->GetMaxFaceInfluences(
                                   pIB,
                                   pMeshContainer->MeshData.pMesh->GetNumFaces(),
                                   &NumMaxFaceInfl);
        pIB->Release();
```

As long as we have 12 matrices in the device matrix palette, we have enough to render any triangle and thus the mesh can be rendered in hardware. If the above function returned more than twelve (unlikely), then ConvertToIndexedBlendedMesh will be able to approximate the mesh to fit. Therefore, we take either the value returned from the previous function or the value of 12 (whichever is less) and check that the hardware supports it. If not, then we will set the D3DXMESH_SOFTWAREPROCESSING flag when we create the skinned mesh and set the mesh container's UseSoftwareVP Boolean to TRUE so that we know we will need to enable software vertex processing during rendering.

In software vertex processing mode we will always have 256 matrices available, so we also calculate the palette entries we will need on the device. Remember that we will feed this value into the ConvertToIndexedBlendedMesh function and it will subdivide the mesh so that no subset uses more matrices than the NumPaletteEntries we specify. Of course, we want this palette size to be a large as possible (no larger than 256 though) so that the mesh can be grouped into larger subsets.

In the following code we calculate this by taking the minimum of 256 and the number of bones in the mesh. This means that if there are more bones in the mesh than 256, then the mesh will need to be subdivided to some degree so that no subset uses more than this amount. If however, the number of bones is less than 256 then these are all the palette entries we need. For example, if there are 128 bones in the mesh, then we will only need at most 128 matrix palette entries to achieve maximum batching efficiency (forgetting about the vertex normals issue for the moment). The number of palette entries is also stored in the mesh container's NumPaletteEntries member because we will need this later during rendering to know how many elements will be in every subset's BoneId array.

```
        NumMaxFaceInfl = min(NumMaxFaceInfl, 12);

        if (m_d3dCaps.MaxVertexBlendMatrixIndex + 1 < NumMaxFaceInfl){
            // HW does not support indexed vertex blending. Use SW instead
            pMeshContainer->NumPaletteEntries = min( 256,
                                        pMeshContainer->pSkinInfo->GetNumBones());
            pMeshContainer->UseSoftwareVP = true;
            Flags |= D3DXMESH_SOFTWAREPROCESSING;
        }
```

The code above shows what happens when the device cannot support the maximum number of matrices that influence a single triangle in the mesh. The result is that we will need to switch over to software vertex processing at the appropriate time.

The following code demonstrates what happens when the hardware can support enough matrices to make sure that at least every triangle is rendered. Notice that in the hardware case we calculate the number of matrix palette entries we are going to use differently from the software case. This time we set the number of entries to either the number of bones in the mesh, or half the number of matrices the hardware can handle, whichever is less. The reason we divide the number of matrices the device can support by two is because we are assuming that the skin has normals. When the *hardware* is performing indexed skinning and the mesh has normals, half of the available matrix slots are reserved for vertex normal transformations. If the number of bones in the mesh is less than half the number of matrices supported by the device, then we can use this value since there is adequate space in the palette. If you are setting up a skin that does not contain normals, then the division by two will not be necessary and you will be able to use all of the matrix slots available on the device.

```
        else{
            pMeshContainer->NumPaletteEntries =
                                min((m_d3dCaps.MaxVertexBlendMatrixIndex + 1) /2,
                                pMeshContainer->pSkinInfo->GetNumBones() );
            pMeshContainer->UseSoftwareVP = false;
            Flags |= D3DXMESH_MANAGED;
        }
```

At this point, we have everything we need to create the indexed blended mesh. The following code shows the call to ConvertToIndexedBlendedMesh which creates a skinned mesh that has both weights and indices stored at the vertex level. It also returns the bone combination table describing which subsets use which matrices.

```
        ID3DXMesh * pSourceMesh = pMeshContainer->MeshData.pMesh;
        pMeshContainer->MeshData.pMesh=NULL;
        pMeshContainer->pSkinInfo->ConvertToIndexedBlendedMesh
                                            (
                                             pSourceMesh,
                                             Flags,
                                             pMeshContainer->NumPaletteEntries,
                                             pMeshContainer->pAdjacency,
                                             NULL, NULL, NULL,
                                            &pMeshContainer->NumInfl,
                                            &pMeshContainer->NumAttributeGroups,
                                            &pMeshContainer->pBoneCombinationBuf,
                                            &pMeshContainer->MeshData.pMesh
                                            );
       pSourceMesh->Release();
    } // end if skin
} //end function
```

When this function returns, we will have a mesh stored in our hierarchy that is ready to be rendered using the pipeline indexed skinning techniques discussed previously.

11.9.5 ID3DXSkinInfo::ConvertToIndexedBlendedMesh

Let us finish off our discussion on creating indexed skinned meshes by looking at the ConvertToIndexedBlendedMesh method of the ID3DXSkinInfo object. This function is very similar to the ConvertToBlendedMesh function used to create the non-indexed skinned mesh in the previous section. The exception is one extra parameter where we specify to the function the number of matrix palette entries we have available for our device. This allows the function to manipulate the mesh such that no single subset uses more matrix influences than this.

```
HRESULT ConvertToIndexedBlendedMesh(

    LPD3DXMESH       pMesh,
    DWORD            Options,
    DWORD            paletteSize,
    CONST LPDWORD    pAdjacencyIn,
    LPDWORD          pAdjacencyOut,
    DWORD           *pFaceRemap,
    LPD3DXBUFFER    *ppVertexRemap,
    DWORD           *pMaxVertexInfl,
    DWORD           *pNumBoneCombinations,
    LPD3DXBUFFER    *ppBoneCombinationTable,
    LPD3DXMESH      *ppMesh
);
```

All of the parameters are the same as the ConvertToBlendedMesh call with the exception of the third parameter (paletteSize). Please refer back to our earlier discussions if you need a refresher on these other parameters.

DWORD paletteSize
This is the number of world matrices that are available for use on the current device. If we were to pass in a value of 64 for this parameter, then this function would subdivide the mesh into different subsets such that no single subset needs to be rendered using more than 64 matrices.

We have now covered everything needed to load and store the indexed skinned mesh. With the exception of one or two places, the code did not change much from the non-indexed case. When D3DXLoadMeshHierarchyFromX returns, the hierarchy will be loaded and will contain all of our indexed skinned meshes in their respective mesh containers.

Once again you are reminded that animating the hierarchy is identical in all of the skinning cases we have covered. We will call the animation controller's AdvanceTime method and then traverse the hierarchy to build the absolute bone matrices for each frame.

11.9.6 Transforming and Rendering the Skin

When we are ready to render our indexed skinned mesh we take the same approach as always – we traverse the hierarchy looking for mesh containers and call their drawing functions as they are encountered. This part is no different than what we have seen in the previous sections. So let us now look at what the DrawMeshContainer method would look like in the indexed skinning technique and then our work is done. This version of DrawMeshContainer is actually the smallest of all the techniques studied.

The first thing that this function does is cast the passed mesh container and frame to our derived class formats.

```
void DrawMeshContainer (LPD3DXMESHCONTAINER pMeshContainerBase,
                        LPD3DXFRAME pFrameBase)
{
    D3DXMESHCONTAINER_DERIVED*pMeshContainer=
                                (D3DXMESHCONTAINER_DERIVED*)pMeshContainerBase;

    D3DXFRAME_MATRIX *pFrame = (D3DXFRAME_MATRIX*)pFrameBase;

    UINT NumBlend;
    UINT iAttrib;
    LPD3DXBONECOMBINATION pBoneComb;
    UINT iMatrixIndex;
    D3DXMATRIXA16 matTemp;
    IDirect3DDevice9 * pDevice;
    pMeshContainer->MeshData.pMesh->GetDevice(&pDevice);
```

After retrieving a copy of our device, we check to see if the mesh stored in this mesh container is a skinned mesh. If so, then we check to see if the mesh needs to be rendered using software vertex processing and enable it if necessary.

```
    if (pMeshContainer->pSkinInfo != NULL)
    {
            if (pMeshContainer->UseSoftwareVP)
            {
                pDevice->SetSoftwareVertexProcessing(TRUE);
            }
```

Next we will set the D3DRS_VERTEXBLEND render state so that we inform the pipeline about how many weights our vertices will have. This tells the pipeline that the N+1 weight in our vertex structure is where the matrix indices are stored. We have this information stored in our mesh container's NumInfl member because it was returned by ConvertToIndexedBlendedMesh. If NumInfl is set to 1 then it means that although we wish to perform indexed blending, there are no weights in the vertices. In this case each vertex will be influenced by one matrix with an assumed weight of 1.0. Every vertex in a subset may still be influenced by a different matrix in the palette, even in this case when the vertex does not have weights (although it would still have the weight being used as matrix indices); we are still performing

indexed matrix transformations and a given triangle could be transformed by up to three matrices (one per vertex).

```
      if (pMeshContainer->NumInfl == 1)
          pDevice->SetRenderState(D3DRS_VERTEXBLEND, D3DVBF_0WEIGHTS);
```

If NumInfl is not 1 then we can just subtract one from it to tell us the number of weights that our vertices use (remember that the number of weights is always one less than the number of influences). So for any cases other than NumInfl = 1 we can use this fact to map directly to the enumerated type for vertex blending.

```
      else
          pDevice->SetRenderState(D3DRS_VERTEXBLEND, pMeshContainer->NumInfl - 1);
```

As a reminder, the flags of the D3DVERTEXBLENDFLAGS enumerated type are assigned the following values:

```
D3DVBF_DISABLE = 0,
D3DVBF_1WEIGHTS = 1,
D3DVBF_2WEIGHTS = 2,
D3DVBF_3WEIGHTS = 3,
D3DVBF_TWEENING = 255,
D3DVBF_0WEIGHTS = 256
```

Having set the D3DRS_VERTEXBLEND render state to correctly describe our vertex format and the weights we wish to use for blending, our next job is to inform the pipeline that we wish to perform indexed vertex blending instead of the standard non-indexed blending. We do this using a device render state that we have not yet seen, called D3DRS_INDEXEDVERTEXBLENDENABLE. It will be set to TRUE or FALSE to enable or disable indexed vertex blending, respectively. So before we render our skin, we must enable it:

```
     pDevice->SetRenderState(D3DRS_INDEXEDVERTEXBLENDENABLE, TRUE);
```

Next, we get a pointer to the mesh container's bone combination table which will contain all of the bone indices used by each subset.

```
     pBoneComb = ( LPD3DXBONECOMBINATION)
                 (pMeshContainer->pBoneCombinationBuf->GetBufferPointer());
```

For each subset we loop through the BoneId array of its bone combination structure. We have stored in the mesh container the size of the matrix palette being used by this mesh, so this is exactly how large the BoneId array of each subset will be. We fetch each matrix index from the subset's BoneId array and, assuming it does not contain UINT_MAX, this is a matrix slot that will be used by this subset. We use the index to fetch the bone matrix from the mesh container and combine it with its corresponding bone offset matrix and set the resulting matrix at its correct slot in the device matrix palette.

```
      for (iAttrib = 0; iAttrib < pMeshContainer->NumAttributeGroups; iAttrib++)
      {
         for ( iPaletteEntry = 0;
```

```
                iPaletteEntry < pMeshContainer->NumPaletteEntries;
            ++iPaletteEntry  )
        {
             iMatrixIndex = pBoneComb[iAttrib].BoneId[iPaletteEntry];
             if (iMatrixIndex != UINT_MAX)
             {
                 D3DXMatrixMultiply( &matTemp,
                              &pMeshContainer->pBoneOffsetMatrices[iMatrixIndex],
                               pMeshContainer->ppBoneMatrixPtrs[iMatrixIndex] );

                 pDevice->SetTransform( D3DTS_WORLDMATRIX( iPaletteEntry ),
                                        &matTemp );
             }
        }
```

By the time we have exited the iPaletteEntry loop above, we will have set all of the matrices used by the current subset in their proper device palette slots. This might be only two matrices or might be over two hundred depending on how many triangles are in the subset and how many matrices need to be used to transform them.

We can now set the textures and material used by the subset (not shown here for simplicity).

```
       //Set Textures and Material of subset here
```

Finally, we render the current subset with the correct palette of matrices that we have just set up.

```
        pMeshContainer->MeshData.pMesh->DrawSubset( iAttrib );

    } // end current attribute 'iAttrib'

  } // end If skin Info != NULL

  else // this is just a regular mesh so draw normally
  {
       pDevice->SetTransform(D3DTS_WORLD, &pFrame->mtxCombined);
       DrawMesh ( pMeshContainer->MeshData.pMesh)
  }

  pDevice->Release();

} // end function
```

As with all other versions of this function, the last section of code is executed if the mesh stored in this mesh container is not a skinned mesh. In that case it is rendered normally, as discussed in previous lessons.

11.9.7 Indexed Skinning Summary

Indexed Skinning Advantages

- Significantly faster than all other skinning techniques because effective batch rendering is maintained when the hardware has a sizable matrix palette.
- Supported in software vertex processing mode if no hardware support exists.
- Possible 256 matrix influences per subset and 12 influences per triangle.

Indexed Skinning Disadvantages

- Not as widely supported as non-indexed, even on fairly recent 3D hardware.
- Still limited to a maximum of 4 matrix influences per vertex, whereas the software skinning technique has no limitation.

Conclusion

We have now covered all three skinning techniques in isolation and have even looked at a lot of basic code to implement them. For a complete implementation description you should now turn to your workbook that accompanies this lesson. It will integrate all three skinning methods into a single mesh class that will make for a nice plug-in to our application framework.

We are not quite finished with our skinning discussions just yet. Our focus in this lesson was almost entirely on how to load and store skins that were pre-generated and stored in X files. In the next chapter, we will use many of the ideas we learned about in this lesson to do things a little differently -- we will create skins, skeletons, and even animations all procedurally. The result will be a solid understanding of how the entire process works, and from a practical perspective, a nice tree class that we can use to liven up our outdoor scenes. Make sure you are comfortable with all of the code we cover in the workbook for this chapter as it will be expected that you understand it when you work on the lab projects in the next chapter.

Workbook Eleven: Skinned Meshes and Skeletal Animation I

Introduction

In the previous lab project we extended our CActor class so that it was possible for an application to use its interface to apply animations to the frame hierarchy. Furthermore, we implemented a system such that these animated actors hierarchies could be efficiently instanced. In this lab project, we will extend CActor to provide automated transformation and rendering support for skinned meshes. Lab Project 11.1 will concentrate on extending the CActor code to support the loading and rendering of skinned meshes using vertex blending and these changes will be demonstrated in an updated viewer application. This will demonstrate our CActor object's ability to play back pre-animated scene hierarchies comprising of both static and skinned meshes. That is, both static and skinned mesh types existing together in a single frame hierarchy.

Lab Project 11.1 – Adding Skinning Support to CActor

You might assume that in order to upgrade our actor code to support the loading and rendering of skinned mesh data, we would need to create a new class; an addition to CActor that is used in special cases for handling skinned mesh data. After all, skinned meshes are processed quite differently from the regular meshes stored in a multi-mesh X file. This is certainly true in the case of actually rendering these entities. However, there is no reason why we cannot extend our actor class elegantly to support both skinned and static mesh hierarchies. This would allow our application to use a single interface for handling both types of mesh data, making the separate tasks usually performed to load and render these two different mesh types invisible to the application.

If you think about how our actor currently works, you should recognize that the changes to the code will not need to be so dramatic. Our actor currently handles the loading and rendering of single or multi mesh (non-skinned) hierarchies. With such a hierarchy, the X file will contain a frame hierarchy which in turn will contain one or more static mesh containers. Each mesh container will contain the data for a single static mesh and will be attached to a single frame in that hierarchy. When this data is loaded from the X file, the data is laid out in the same way inside the actor as we have seen in the previous lab projects.

We know that before we render our actor we must first apply any animations to the relative frame matrices of the hierarchy and then perform a traversal of the hierarchy to update the absolute matrices for each frame. At that point, we are ready to render all the meshes in that hierarchy. The current rendering strategy of our actor similarly involves another traversal of the hierarchy. Currently, when we render our actor, we start at the root frame and traverse the entire hierarchy looking for mesh containers. Whenever we find a frame with a mesh container structure attached, we know that this frame owns a mesh. At this point, we set the frame's absolute matrix as the current world matrix for the D3D device and render the mesh stored there. After we have traversed the hierarchy, all meshes managed by the actor will have been rendered.

So, what needs to be changed in order to render skinned meshes stored in a hierarchy? The first obvious difference is that the absolute frame matrix alone no longer describes the position and orientation of the

skinned mesh within the hierarchy (or world). Instead, the mesh container will contain an ID3DXSkinInfo object which will tell us the connection between vertices in the skin and frames/bones in the hierarchy. More specifically, it will tell us which matrices in the hierarchy should be used to multi-matrix transform (vertex blend) the vertices of each subset of the mesh. Before rendering each subset, we must find the absolute matrices of the frames that influence that subset. We then set these matrices on the device, enable vertex blending, and then render the triangles in that subset. The frames of our hierarchy which influence the transformation of subsets in our skinned mesh are often referred to as the 'bones' of the mesh. The absolute matrices for these frames are referred to as the bone matrices. While this often sounds a bit scary to the uninitiated, we are quite used to working with bones. They are, after all, just frame matrices going by a different name.

At first, it seems that the rendering method of the actor would need to be able to maintain two modes. If it is an actor that contains skinned meshes we should render using the slightly more complex multi-matrix blending method; if not, we render using the single matrix method that our actor has supported for the last few lessons. However, this is not quite the way we want our actor to work. We do not want our actor to be in either one mode or another because it is possible (and not at all uncommon) that we might load an X file (or procedurally build a hierarchy) that contains both static and skinned meshes all within a single hierarchy.

It may be difficult at first not to visualize a hierarchy that contains a skinned mesh, as always being limited to defining the bones (skeletal structure) of a single character or object. It is more intuitive for us to picture an actor as containing a single character bone hierarchy containing a single skinned mesh for which the the hierarchy forms its skeleton. If we were to extend our actor to only work in this way, then we would indeed be able to have an actor that works in one of two modes. However, we would find this extremely limiting and it would actually fail to correctly manage and render a large number of X files and scene configurations which contain both skinned and regular meshes all within the same frame hierarchy.

For example, in lab project 11.1, we show how an X file is loaded into our new viewer application. It is worth remembering that the X file could actually contain an entire scene and not just a single skinned mesh. That is to say, it could contain the static geometry for lots of buildings and multiple skinned meshes (for example). The X file could also contain pre-recorded animation data that when played, make the characters walk about within that scene. This is a single X file, which means it would be managed and rendered by a **single actor**. The actor will have a frame hierarchy that contains not only the frames to position the static meshes in the scene (the rooms, buildings, and tables, etc.) but also, sub-sections of that hierarchy will contain the complete skeletons for two or more character meshes (the skins). While translation keyframes are applied to the skeletons of each character to make them move around the scene independently from each other, they are still attached to the root frame of the scene and share the same frame hierarchy with other static meshes. The character skeletons are now just sub-sections of that entire scene hierarchy. The animation controller defined for this scene would only apply animations to the frames of the hierarchy that comprise the character skeletons. Therefore, the static and skinned meshes are all connected and the entire pre-recorded scene (including the character meshes) could be repositioned somewhere else in world space simply by applying a new world matrix to the actor's root frame.

One thing should be clear from the above discussion. We need to allow our actor's hierarchy to manage both skinned and static meshes and allow them to harmoniously co-exist within the same actor hierarchy. The actor's rendering logic should be extended such that it can recognize and render both types of mesh when doing a render traversal. For example, when traversing the hierarchy during a render pass, it will search for all frames that have a mesh container attached. For each one that is found, it will need to determine whether or not this mesh container stores a skinned or regular mesh and choose a suitable rendering strategy for each case.

As discussed in the textbook, when a frame hierarchy is being loaded and mesh data is attached to a frame (a relationship defined in the X file), the ID3DXAllocateHierarchy::CreateMeshContainer call-back function will be called. This function accepts an ID3DXSkinInfo interface pointer, which we have thus far ignored in previous lab projects. If this pointer is NULL, then it means the mesh currently being loaded is a static mesh and we load it in the same way we have in all previous projects. If this interface pointer is not NULL, then it means there is also skinning information defined for this mesh in the X file. Since the D3DXMESHCONTAINER structure contains an ID3DXSkinInfo pointer member, we can store this ID3DXSkinInfo interface pointer in the mesh container during the hierarchy loading/creation process. We will need it later anyway to figure out which frame matrices in the hierarchy must be used to transform its various subsets. The upshot of this is, during the rendering traversal of the hierarchy, whenever we find a mesh container, we can simply test to see if its ID3DXSkinInfo pointer member is set to NULL. If it is, then it is not a skinned mesh and should be transformed and rendered in the regular way (by simply setting the absolute matrix of the current frame as the device world matrix prior to rendering that mesh). If it is not NULL, then we know we have a skinned mesh and can deploy an entirely new rendering strategy for that mesh. With the latter case, we would use the ID3DXSkinInfo object to determine which matrices to set simultaneously on the device prior to rendering each subset of the skin.

With this type of logic in place, our application can use a single CActor to load any type of scene without having to worry about what that scene contains; it will not have to perform any special processing in the skinned case. The application may use the actor to load an X file which contains a single skinned mesh character (where the hierarchy represents a single character skeleton) for example, or it may use the actor to load an entire scene of static meshes. The actor could also be used to load an X file that contains both a mix of regular and skinned meshes. In every case, the application has no logic to perform; it simply calls CActor::LoadActorFromX and lets the actor handle the details of both loading and rendering that geometry.

An example of a scene that might contain multiple static and skinned meshes might be a forest scene. The basic hills, valleys and buildings comprising the the scene would be arranged hierarchically and would be constructed as static meshes. However, it may also contain a few trees that sway in the wind. We might decide that the trees themselves will be represented as skinned meshes and frames will be added to the hierarchy inside the tree branches to act as the branch bones. The artist might then build animations to rotate and translate those bones in a random pattern so that the branches appear to be swaying in the wind. The actor will handle this automatically and the application will use the actor in exactly the same way as it has in previous projects. The actor will still have a single frame hierarchy and will still apply animations (via its animation controller) to certain frames in that hierarchy. The fact that some of those frames (the ones being animated) are actually being skinned by branch meshes, does not effect the hierarchy representation or the way animations are applied to that hierarchy.

For example, consider a simple example of a winter tree that has a tree trunk and four branches stored in an X file.

Figure 11.1

While this might look like a single tree mesh, assume it consists of five separate meshes. The tree trunk would be one mesh and each of the four branches shooting off of the main trunk might also each be individual meshes. The artist might decide when building this tree that the branches should sway in the wind and therefore might decide to place some frames/bones inside the branches. We will imagine for now that the root frame has been placed inside the trunk. The artist could build subtle animations that would be applied to the branch frames to make them move by some small amount as a result of wind being applied. We might decide however that as the trunk of the tree itself would never actually move, we will not make it a skinned mesh like the four branch meshes. This decision might be made in an attempt to aid rendering performance as rendering a skinned mesh is more computationally expensive. Now, this is a slightly odd example because normally you would make the entire tree a single skinned mesh so that no cracks or gaps appear between the branch meshes and the trunk, but bear with us for the purposes of this example.

We know that when the above tree was exported, we would have an X file that contained a frame hierarchy. In that hierarchy there would exist five meshes -- one static mesh (the tree trunk) and four skinned meshes (the tree branches). Therefore, while the hierarchy described by the X file would essentially contain the bones of the entire tree, not all meshes contained in the hierarchy would need to be multi-matrix transformed when rendered. For the sake of this example, let us imagine the frame hierarchy for the tree was defined as shown below in Figure 11.2.

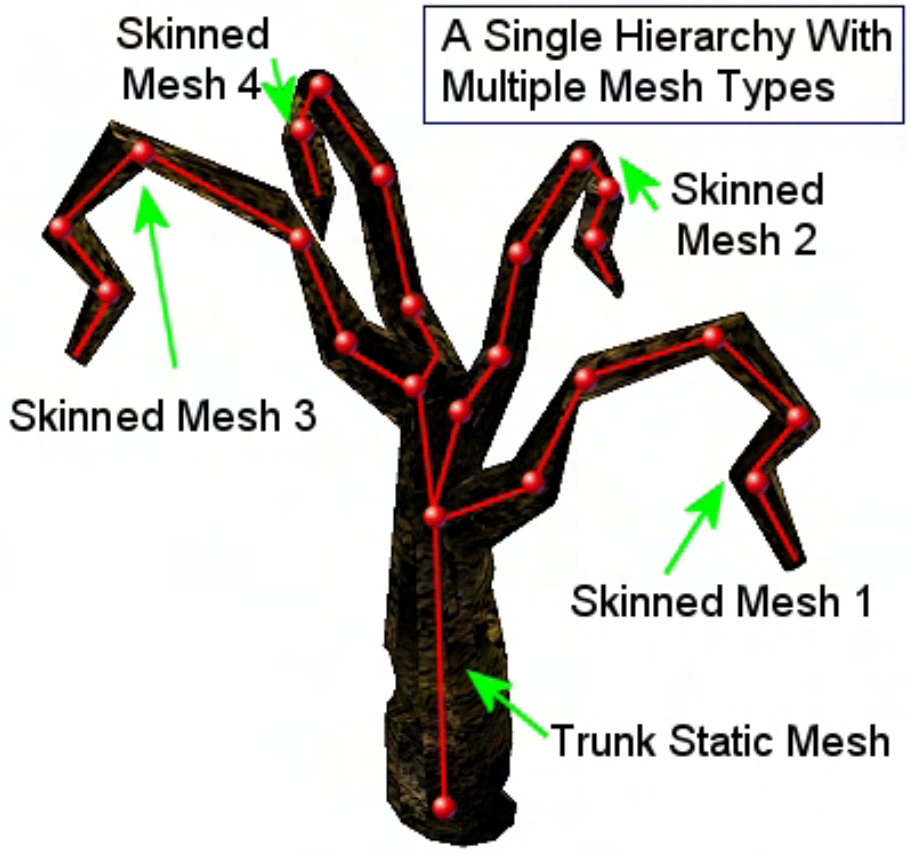


Figure 11.2

The bone (red sphere) at the bottom of the tree trunk is assumed to be the root frame in this diagram and its absolute matrix would be used to directly position the static trunk mesh prior to rendering. We might say then, that this is not a bone at all in the traditional sense. The rest of the frames in the hierarchy are bones as these will be mapped to vertices in the branches such that, when animation data rotates or translates these bones matrices, the branches of the tree will be animated. The root frame however is not a bone in the traditional sense as its absolute matrix is not used directly to transform and render any of the vertices in the branch meshes. That is, we will never set the root frame's absolute matrix while rendering the skins; we only do this when we render the trunk mesh (in which case it will be the only matrix set on the device). However, the root frame is certainly a bone in another sense as any rotations or translations applied to this bone will alter the positions and orientations of all the other bones in the tree also.

Figure 11.3 shows how a tree might be represented as a hierarchy in the X file and inside our actor. Figure 11.3 is not an accurate depiction of the bones used in 11.2, but will serve for the purpose of this example.

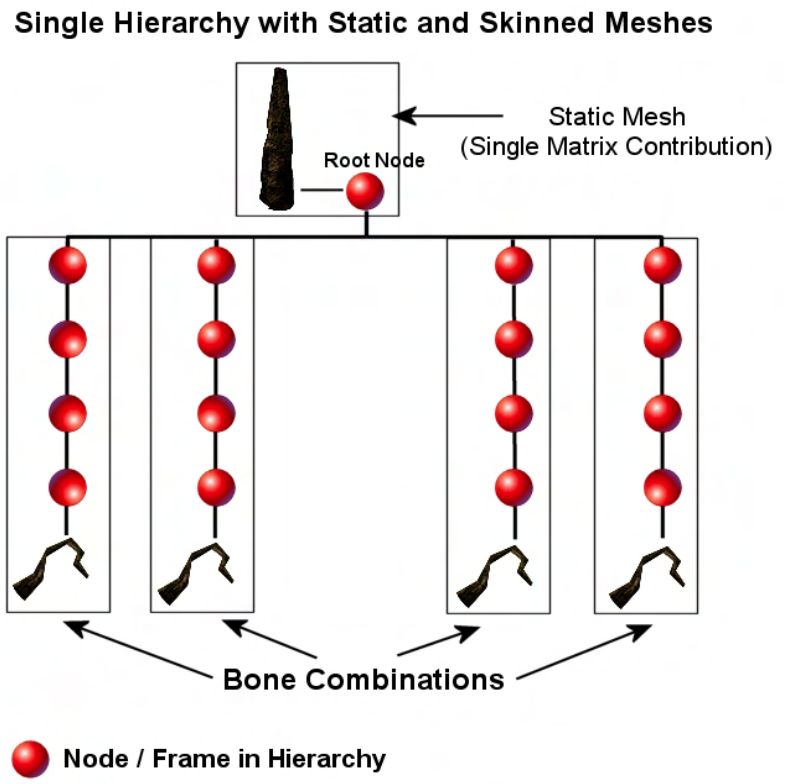


Figure 11.3

In Figure 11.3 we can see how the hierarchy looks with both its static trunk mesh and its four branch skins. In this example, the trunk mesh is in a mesh container attached to the root node. Since this is a static mesh, the absolute matrix of the root node will be used as the world matrix for rendering this static mesh. As children of the root node, there are several frames that will be used as the bones of the branch meshes. The block boxes indicate the frames in the hierarchy that influence each mesh.

Looking at this example, we can see that our actor is using a combination of static and skinned meshes. Let us walk through the rendering strategy, assuming that all animations have been applied and the absolute matrix of each frame has been updated and is now ready for the rendering traversal pass.

Our actor's rendering function would start at the root frame. It would find that there is a mesh container there, but would also discover that this mesh container contains no skinning information. This means it is a static mesh and the absolute matrix stored in the root is the actual world matrix that should be set on the device to correctly transform and render the mesh. So the matrix would be set as the current world matrix on the device and the mesh would be rendered.

We would now discover that this frame has children, so we would then traverse into each of these little sub-hierarchies. We will step through the traversal of the left-most sub-hierarchy first. We would get the first child frame and discover that no mesh container is stored there, so there is nothing to render. We would then step down into its child where the same discovery would be made again. There is still nothing to do so we would step down into the third level of this sub-hierarchy where once again, there is

no mesh data attached and nothing to render. Finally, we would step down into the fourth frame of this sub-hierarchy where the skin (i.e., branch mesh) is actually stored in a mesh container.

At this point we would test to see if this mesh container had skinning information loaded for it and would find that it does indeed have a valid pointer to an ID3DXSkinInfo interface. This means this is a skinned mesh. We would now have to loop through each subset in the mesh and set all the matrices that influence that subset in the device's matrix palette. With all the matrices that influence the subset now bound to the device, we would enable vertex blending and render the subset. In this example, the four matrices that share the black box with the branch mesh (Figure 11.3) would be sent to the device (combined with their respective bone offset matrices) and used to multi-matrix transform the vertices of the subset into world space. For both regular and skinned meshes, we must render every subset. Unlike in the regular case however, when rendering a skinned mesh, each subset may require a different set of bone matrices to be sent to the device before it is rendered. Therefore, when rendering the skinned mesh, we are no longer just concerned with batch rendering by texture and material. We must now also make sure that we also set the correct matrices for each subset that we render, giving us yet another batching key to worry about for efficient rendering.

You will see when we cover the upgraded code for CActor, that each mesh container that stores a skinned mesh will also store a bone combination table. This will be calculated at hierarchy load time (inside the ID3DXAllocateHierarchy::CreateMeshContainer method) and will contain information describing, for each subset in the skin, which matrices must be set on the device in order to render it.

So we now know that our actor should have the ability to render a hierarchy that may contain skinned and regular meshes and we should place no burden on the application to provide any support for this. We want CActor to encapsulate the hierarchy loading, animation, transformation, and rendering just like it always has, including handling the cases of skinned meshes in our scene.

For another more complex example, an X file might contain multiple tree representations like the one previously described. The skeletal structures for each tree could be combined in a single hierarchy as shown in Figure 11.4. In this example the X file contains two trees. Each tree has a static trunk mesh and four branch skins. In this example, each trunk mesh is a child of the root. This shows an example of an X file/hierarcy that contains both multiple static and skinned mesh types.

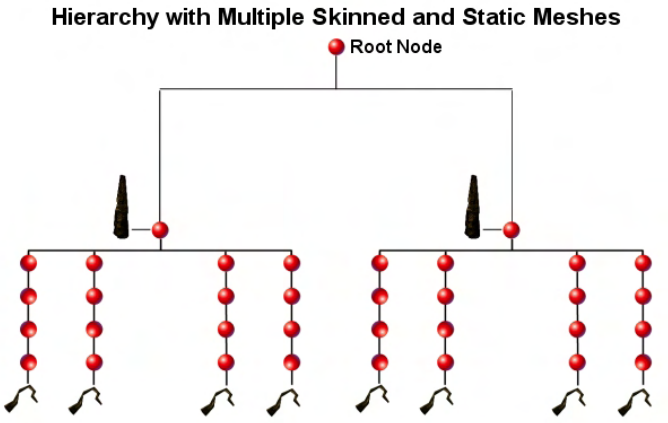


Figure 11.4

Not only should CActor handle skinned mesh support behind the scenes without burdening the application with any significant additional work, it should also support a number of different skinning techniques. We learned in the accompanying textbook that there are three types of skinning techniques we can use when performing character skinning.

We would like our actor to be able to support software skinning. While this is the slowest technique available to us (because the hardware is not utilized for purposes of transformation), it is also the only technique that imposes no limitations on the number of bones which can influence a single vertex, triangle or subset. Therefore, we may sometimes find that if we wish to accurately render a skinned mesh with an extremely complex skeletal structure and maintain its integrity to the best of our ability, we may sacrifice speed for the ability to render an object which has many bone contributions per vertex. When using hardware techniques, we can never have more than four bone influences per vertex, so software skinning support within CActor may come in handy. While it is extremely rare in real-time work that you would ever need to render geometry that uses more than four bones contributions per vertex, let us not shy away from providing that ability. We never know when we might need it.

We will also want CActor to support two types of hardware skinning that is available on most DirectX 9 graphics cards on the market today. Non-indexed skinning is usually the least desirable of the hardware techniques available because, while the hardware is utilized for the multi-matrix vertex transformations, this speed savings can be offset (sometimes) by the fact that we can only set four matrices on the device at any one time. This usually results in the source mesh being divided into many small subsets such that no single subset is influenced by more than four bones (even if the triangles share the same texture and material). This is a shame, since we know that we could usually render all triangles that share the same textures and material with single draw call. This need to render based on bone contribution creates smaller subsets and can reduce batch rendering performance.

One of the saving graces of the non-indexed case is that we can at least optimize the rendering a little by making sure that when we render a subset, we only use the weights and matrices that the subset actually needs. For example, if we render subset A which has four bone contributions, we can set the D3DRS_VERTEXBLEND renderstate such that the three weights (the fourth is created on the fly by the pipeline) are used to weight the contributions of the matrices in matrix slots 1-4 on the device. However, if subset B is only influenced by two matrices, we can adjust the D3DRS_VERTEXBLEND render state so that only one weight (the second weight is generated on the fly) is used to weight the matrix contributions of matrices in device slots 1-2 only. This way we do not waste time transforming vertices by four matrices regardless of whether a subset is actually influenced by all four of those matrices. This is a worthwhile optimization since all of our vertices will share the same vertex format. If one vertex in the mesh is influenced by four bones, every vertex in the mesh will be transformed by four weights (with space for three in its vertex structure). If a given subset is only influenced by two bones, the remaining two unused weights in the vertex structure will be set to zero, and any matrices that are assigned to slots 3 and 4 on the device will have their contribution zeroed out when the vertex is transformed. The problem with this is that without trimming the number of weights used on a per-subset basis (using the D3DRS_VERTEXBLEND renderstate), the vertices would still be transformed by all four matrices even though the weights of the vertex would zero out the contributions of the final two matrices in this example (making them unnecessary). We never want to be performing unnecessary vertex/matrix multiplications on a per-vertex level since this is very inefficient.

We also want our CActor to support hardware indexed skinning.This same optimization cannot be performed in the indexed skinning case, because the vertex weight containing the indices will always be in the same position in each vertex. Therefore, if we have an indexed skin where one vertex is influenced by four bones, the first three weights (the fourth weight is created on the fly) will always be used to weight the contributions of four matrices. The fourth weight in the vertex structure is used to hold the four matrix indices. In this example, even if a subset is only influenced by two bones (which would only require a single vertex weight in the non-indexed case), the indices will still be stored in the fourth vertex weight and the pipeline will need access to them. So we must set the D3DRS_VERTEXBLEND render state so that all weights (up to the weight that stores the indices) are used. This may cause vertex-matrix multiplications to occur unnecessarily for weights that are set to zero. Of course, this downside is often a worthy tradeoff for the fact that in the indexed skinning case, we have a much bigger matrix palette at our disposal and (potentially) the ability to have a single subset influenced by 256 matrices without having to be subdivided. The result is fewer subsets in the indexed skin and thus greatly enhanced batch rendering performance.

So it seems that our actor will need code to load and render static and skinned meshes simultaneously, and it must also have multiple paths of code for rendering skinned meshes using the three skinning techniques available. The code to load, store and render software skins, non-indexed skins and indexed skins is all slightly different and our actor will need to implement all paths.

Intelligent Auto-Detection

Not only will our actor support the three unique flavors of skinning, it should also provide a means for the application to specify which skinning technique is preferred. Of course, while it is definitely necessary in some circumstances for the application to specifically state which skinning technique should be used, in the general case, most applications would simply prefer the most efficient one to be chosen given the limitations of the hardware it is running on. So we can implement a method that would allow the application to specify the type of skinning it would like the actor to use and we will certainly do that. However, we will not stop there. We do not want to place the burden on the application to determine which skinning techniques are available on the current system before informing the actor of its choice. Therefore, we will also implement a useful auto-detection mode that will allow the application to let the actor determine the best skinning technique to use, based on the installed hardware. The application can simply load the X file and let the actor perform the hardware capability tests before choosing an appropriate skinning method. This maintains the actor's plug and play design goal. We wish to use CActor in any application without excess burden placed on the application. As far as the application is concerned, our actor should just be an object in the world that can be animated and rendered each frame (via its interface). The fact that the actor is internally a complex hierarchy of both regular and skinned meshes should not pollute the application code.

In addition to implementing an auto-detection feature in our actor (as well as a means to allow the application to specifically state which skinning mode the actor should use if it wishes), we will also implement a 'hint' system that the application may use to help bias the auto-detection mechanism of the actor. This will help it make the correct choice on the application's behalf.

Without any hints being passed to the actor by the application, the auto-detection mechanism will make the logical choice under most circumstances. If the hardware can support a mesh using indexed skinning, then this will be chosen and the loaded mesh will be converted into an indexed skin. As we know, this is usually the optimal choice. If hardware support for indexed skinning fails, then the actor will test the hardware for non-indexed skinning support. If this is supported then this will be the next best solution. In other words, if we have hardware support for both indexed and non-indexed skinning solutions for the mesh, the auto-detection mechanism will always choose indexed over non-indexed.

There may be times however when you wish to change this behavior. You may decide that you would like the actor to choose non-indexed hardware skinning over hardware indexed skinning if both are supported. Therefore, we will have several 'hint' flags that the application can pass to the actor to suggest this preference. While it may seem as if you would never want to do such a thing, there are times when non-indexed skinning can actually outperform indexed skinning due to the elimination of the zero contribution weights during rendering. Also, it may be the case that the hardware incurs a slight performance hit when performing indexed skinning due to the additional lookup into the matrix palette. You will usually find however, that the larger matrix palette and subsets resulting from indexed skinning allow it to outperform the non-indexed method.

Finally, what should our auto-detection method do if hardware support for both indexed and non-indexed skinning is absent? You might think that we would fall back to the pure software skinning mode in this case, but that is not what we will do. In fact, the auto-detection mechanism of CActor will never fall back to using the pure software skinning technique by itself. If this mode is required, it must be specifically asked for by the application. What the auto-detection mechanism will do when no hardware support for skinning is available, is use software vertex processing versions of the pipeline's skinning methods. Remember, just because the hardware does not support indexed skinning, does not mean that we cannot use DirectX's skinning pipeline. It simply means we will need to enable software vertex processing when processing this mesh such that the vertices of the mesh are transformed in software (using the host CPU) instead of by the hardware. DirectX's indexed and non-indexed skinning methods will always be available to us even if hardware support for skinning is unavailable. Therefore, what our auto-detection mechanism will do in the absence of any hardware skinning methods, is choose the pipeline's non-indexed skinning method using software vertex processing. We have generally found in our own lab tests that software non-indexed skinning is faster than software indexed skinning in a variety of cases. This is likely due to the fact that the redundant matrix multiplications with zero weight vertices in the indexed case incur a much higher performance hit now that the matrix multiplication is being done on the host CPU. However this is quite dependant on the model and as such, is not always the case. Therefore, the application will also be able to instruct the actor to choose software indexed skinning first (if desired) through the use of 'hint' flags to the auto-detection system.

We now have a path that will always locate an applicable skinning technique even if there is no hardware support. Once again though, even in the software case, hint flags can be used to bias the auto-detection process. We will have a flag that will instruct the actor that if no hardware support is available, we would prefer to use indexed skinning with software vertex processing rather than non-indexed skinning with software vertex processing (the default software technique that is used if no hardware is available).

The CActor class now has a new enumerated type that contains a number of flags. The application can pass one or more of these flags into the CActor::SetSkinningMethod member function prior to loading the actor (via CActor::LoadActorFromX). The SetSkinningMethod function simply stores the passed flags in a new actor member variable so that the LoadActorFromX method will be able to access this information when deciding what type of skin to create.

Below you can see the new SkinMethod enumeration which is now part of the CActor namespace. It contains a number of flags that can be passed into the SetSkinningMethod member function to describe how we would like the meshes to be built during the loading process.

```
enum SKINMETHOD    { SKINMETHOD_INDEXED    = 1,
                     SKINMETHOD_NONINDEXED = 2,
                     SKINMETHOD_SOFTWARE   = 3,
                     SKINMETHOD_AUTODETECT = 4,
                     SKINMETHOD_PREFER_HW_NONINDEXED = 8,
                    SKINMETHOD_PREFER_SW_INDEXED = 16 };
```

Below is a description of the various flags and how they will influence the auto-detection mechanism of the actor during loading.

SKINMETHOD_INDEXED
This flag is tells the actor that we wish to use the pipeline's indexed skinning method. This flag disables the auto-detection process. If hardware support for indexed skinning is not available, then the pipeline's software vertex processing pipeline will be used to perform the indexed skinning. This flag is mutually exclusive to all other flags.

SKINMETHOD_NONINDEXED
This flag is tells the actor that we wish to use the pipeline's non-indexed skinning method. This flag disables the auto-detection process. If hardware support for non-indexed skinning is not available, then the pipeline's software vertex processing pipeline will be used to perform non-indexed skinning. This flag is mutually exclusive to all other flags.

SKINMETHOD_SOFTWARE
Specifying this flag will disable the actor's auto-detection mechanism and tells it *not* to use the DirectX pipeline's skinning methods and to prefer the full software skinning technique instead. Specifying this flag is the only way the actor will ever use a full software skinning implementation. When in pure software mode, the DirectX pipeline will not be used in any way to transform the vertices of the skin into world space. Instead, we will perform the multi-matrix transformation of vertices from model space into world space ourselves. Rendering this mesh will require us setting the world matrix to an identity matrix since we will be passing DirectX a world space transformed skin. This is all done inside the actor's rendering code.

This is the slowest skinning method but is useful if you do not wish to have any limitations placed on the geometry. Even when using software vertex processing with DirectX's skinning methods, we are still limited to a maximum of four influences per vertex. With a pure software implementation this is not the case. We are not limited by either the number of bones that influence a vertex or the number of matrices that can be used in total. This flag is mutually exclusive to all other flags.

SKINMETHOD_AUTODETECT
This is the auto-detection mode flag. It can be combined with the final two modifier flags (shown below). This flag cannot be combined with any of the three flags described previously.

This flag places the actor into auto-detection mode; the default mode of the actor. When a mesh is loaded the actor will query the capabilities of the hardware and choose (what it considers to be) the most appropriate method. It will first try to use hardware indexed skinning. If not available, it will try to use hardware non-indexed skinning as the next best option. Finally, if hardware non-indexed skinning is not available, it means there is no hardware support. It will then fall back to using the pipeline's non-indexed skinning technique with software vertex processing.

This is the default behavior of the auto-detection process. However, you can combine any of the following flags with the SKINMETHOD_AUTODETECT flag to alter the way in which the auto-detection mechanism chooses it skinning technique:

SKINMETHOD_PREFER_HW_NONINDEXED
This is a hint flag that can be combined with SKINMETHOD_AUTODETECT. It can also be combined with SKINMETHOD_PREFER_SW_INDEXED. It changes the default auto-detection process of finding a suitable hardware method such that the actor will first favor non-indexed hardware skinning over indexed hardware skinning if both are supported. You may find with some models that non-indexed skinning may actually be faster than indexed skinning. This hint flag gives the application control in these circumstances. If this hint flag is not used, indexed skinning will always be favored in hardware.

SKINMETHOD_PREFER_SW_INDEXED
This is another hint flag that can be combined with SKINMETHOD_AUTODETECT. It can also be combined with SKINMETHOD_PREFER_HW_NONINDEXED. It changes the default auto-protection process for finding a suitable software skinning method if no hardware support is found. Be default, if no hardware support for skinning is available, it will choose the pipeline's non-indexed skinning method using software vertex processing. This flag instructs the actor, in the absence of hardware support, to favor the pipeline's indexed skinning method with software vertex processing.

These flags cover all options and provide our actor with either explicit instructions about which skinning method to use, or asks for the auto-detection mode to be used along with hint flags that can be used to further control the process. This means our actor will always be able to find a suitable skinning technique without burdening the application with any detection code. For example, in lab project 11.1, the following code is now used to load the actor:

```
CActor * pNewActor = new CActor;

pNewActor->RegisterCallback(  CActor::CALLBACK_ATTRIBUTEID,
                              CollectAttributeID, this );

pNewActor->LoadActorFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

As you can see, nothing has changed at all. We simply allocate a new actor, set the callback method to handle the loading and storing of texture and material resources (optional), and then load the actor just like we always have. We have not even set the skinning method of the actor. So how does the actor know which method to use when loading in skinned data from the X file?

It just so happens that the skinning method of the actor is set to SKINMETHOD_AUTODETECT by default in the constructor. Therefore, since this is the mode you will probably use for your actor object's most of the time, we do not even have to bother setting it. The same code our application used to load a simple mesh hierarchy is now being used to load a hierarchy with (potentially) multiple regular and skinned meshes and animation data. Note how our changes to the actor class have not affected the way the application uses it. Our actor will even been rendered in exactly the same way as previous lab projects since it will be the actor's internal rendering code that will handle the detection and rendering of any skinned meshes contained within.

While the above code is probably how your application will most likely use the actor, it does not demonstrate the various flags we just discussed. If you wish to change the skinning method used by the actor, or bias its auto-detection mechanism from the standard path in some way, you must use these flags and the CActor::SetSkinningMethod function to inform the actor *prior to* loading the X file. Let us have a look at some examples that we might use:

Example 1

```
CActor * pNewActor = new CActor;

pNewActor->RegisterCallback(  CActor::CALLBACK_ATTRIBUTEID,
                              CollectAttributeID, this );

pNewActor->SetSkinningMethod ( CActor::SKINMETHOD_INDEXED );

pNewActor->LoadActorFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

In this example, after creating the actor and registering its attribute callback function, we use the SetSkinningMethod to inform the actor that we definitely wish to use indexed skinning. When the actor is loaded, if hardware support for indexed skinning is available, it will be chosen. However, if no hardware support is available then the pipeline's indexed skinning technique will still be used but with software vertex processing. Even if hardware support for non-indexed skinning is available, it will not be chosen.

Example 2

```
CActor * pNewActor = new CActor;

pNewActor->RegisterCallback(  CActor::CALLBACK_ATTRIBUTEID,
                              CollectAttributeID, this );

pNewActor->SetSkinningMethod ( CActor::SKINMETHOD_SOFTWARE );

pNewActor->LoadActorFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

This example shows the only way the application can instruct the actor to completely bypass the DirectX pipeline's skinning techniques and use a pure software implementation. This can be used for special cases where the DirectX skinning techniques would not suffice (e.g. you wish to use a model that has more than four influences per vertex, and do not want it downgraded to fit the pipeline's techniques).

This is the slowest mode when skinned meshes are contained in the actor hierarchy, because all transformations will happen in software.

Example 3

```
CActor * pNewActor = new CActor;

pNewActor->RegisterCallback(  CActor::CALLBACK_ATTRIBUTEID,
                              CollectAttributeID, this );

pNewActor->SetSkinningMethod ( CActor::SKINMETHOD_AUTODETECT |
                               CActor::SKINMETHOD_PREFER_HW_NONINDEXED |
                               CActor::SKINMETHOD_PREFER_SW_INDEXED );

pNewActor->LoadActorFromX( strFileName, D3DXMESH_MANAGED, m_pD3DDevice );
```

In this example we chose the actor's auto-detection mechanism for finding a suitable skinning method. However, rather than the default mode, we also specify two hint flags. With these two flags the auto-detection mechanism will always try to find a hardware skinning solution first. However, if both indexed and non-indexed skinning techniques are available, it will choose non-indexed hardware skinning. The second hint flag will be ignored if a hardware solution is found and will only influence the auto-detection process when searching for a software solution. The normal strategy of the auto-detection process when no hardware solution is found is to use the pipeline's non-indexed skinning technique with software vertex processing. This flag will instruct the actor to use the pipeline's indexed skinning technique with software vertex processing instead.

We now understand how we wish the auto-detection method of the actor to work. This will go a long way towards helping us understand the code when we examine it. However, before we finally delve into the CActor code, we should discuss how textures, materials, and other attribute based properties will be handled for skinned meshes within the hierarchy.

Skinned Meshes and Subset IDs

In our previous implementations of CActor, each mesh (a CTriMesh) in the hierarchy had the ability to be in either managed or non-managed mode. In managed mode, each mesh contained its own list of texture pointers and materials, and the attribute IDs of each triangle in the attribute buffer indexed into this local list. This meant that attribute IDs were local to the mesh. If the mesh had 5 subsets, the attribute IDs of each triangle have values between 0 and 4. In this mode, the mesh is self-rendering since it has everything it needs stored internally. The mesh renders itself by looping through each of its subsets, setting the texture and material in the matching position in its attribute array before rendering the subset.

This causes a slight problem when dealing with skinned meshes because after we have converted the mesh into a skin (using either ConvertToBlendedMesh or ConvertToIndexBlendedMesh) the resulting mesh will contain completely different subsets from the original mesh that was loaded. The mesh may have been subdivided into additional smaller subsets based on bone influence. The problem is that the

original per-subset attribute information is loaded by D3DX with a 1:1 mapping with subset IDs. We know that the first texture and material in the mesh's internal attribute array (texture and material table) is the texture and material for the first subset, and so on. After mesh conversion, we would now have an array of attributes stored in the mesh but we can no longer use the new subset IDs of the returned skinned mesh to index into this attribute table. The 1:1 mapping has been lost.

Imagine that subset 0 in the original mesh (which uses texture[0] and material[0]) was influenced by 8 bones and we are using non-indexed skinning and a managed mode mesh. We know that the maximum number of influences per subset in this case can be four. When we use the ID3DXSkinInfo::ConvertToBlendedMesh function to convert the original mesh to a skin, it would have detected this fact and would have split this subset into two in the resulting mesh. Therefore, what was originally a single subset with an ID of 0, has now been split into two subsets with IDs of 0 and 1 respectively. The attribute IDs of all triangles that were in the **original** second subset (assuming one existed) have been incremented so that they now belong to the third subset, and so on. This is done so that the two new subsets that have replaced the initial subset can be rendered one at a time using four matrices each, which the pipeline can handle.

Now we see the problem. Subsets 0 and 1 in the new mesh originally belonged to the same subset (subset 0) so they both must use texture[0] and material[0] in the mesh's attribute array when rendered. The problem is, in managed mode, we always used a 1:1 mapping between subset IDs and attributes in this array in our rendering logic, and this has now been compromised. To render subset 1 in the new mesh, we must still use texture[0] and material[0] and not texture[1] and material[1]. Of course, this is a simple example but in a more complex case, all the original subsets may have been divided into multiple subsets in the resulting mesh. This would mean that all of the attribute IDs of every triangle in the attribute buffer will have been changed. It would seem as if there is no longer a way for us to determine which attribute should be used with which subset.

Of course, this is not the case, because if it were, converting a mesh to a skin would be useless. As an example of how we must overcome this problem, let us imagine that we get passed a mesh in the CreateMeshContainer method that contains skinning information. Let us also imagine that this mesh originally has five subsets. Because this is a skinned mesh we will need to convert the mesh into a skin by using either the ConvertToBlendedMesh function or the ConvertToIndexBlendedMesh function depending on whether we wish to create an indexed or non-indexed skin.

Assuming we are in managed mode for this example, the first thing we might do is load the textures (using the callback) and materials and store them in the mesh's attribute array. At this point all is well because the attribute IDs of each triangle simply index into this texture and material array we have generated inside the mesh. So before we render subset[4], we would need to set texture[4] and material[4]. However, our final task will be to convert this mesh to a skin using ID3DXSkinInfo::ConvertToBlendedMesh. At this point a new mesh will be created and the original one can be released. Let us also assume that the ConvertToBlendedMesh function decided to subdivide every subset in the original mesh into two unique subsets in the resulting mesh for reasons of coping with the maximum bone influences. In the new mesh we now have ten subsets instead of five. The subset IDs of this mesh will now range from 0 – 10 even though our texture and material array still ranges from 0 – 5.

All would be lost for us at this point were it not for the fact that ConvertToBlendedMesh returns an array of D3DXBONECOMBINATION structures. There will be one element in this array for each subset in the **new** skinned mesh. In our example this means there will be 10 elements in this array. Each element in this array contains the rendering information for each subset in the new mesh. It contains an array of bone matrix indices which must be set on the device prior to rendering the subset and also provides useful information such as where the vertices and faces in this subset start and end in the vertex and index buffers respectively. This allows us to efficiently transform and render only the sections of the vertex and index buffer that is necessary when rendering a subset. Finally, and most important to this discussion, is that each D3DXBONECOMBINATION structure also contains the original subset ID that this new subset was created from. As we originally had a 1:1 mapping between the subset IDs and the attributes in the attribute buffer, this original subset ID member tells us exactly where the texture and material we need to use to render this new subset is stored in the mesh attribute array

```
typedef struct _D3DXBONECOMBINATION {
    DWORD AttribId;
    DWORD FaceStart;
    DWORD FaceCount;
    DWORD VertexStart;
    DWORD VertexCount;
    DWORD *BoneId;
} D3DXBONECOMBINATION, *LPD3DXBONECOMBINATION;
```

The AttribId member is the key. It tells us the original index into the attribute array (the array of textures and materials) that should be used to render this subset. If the original first subset (subset 0) was divided into two unique subsets in the skinned mesh, the AttribId member of the first two D3DXBONECOMBINATION structures will be set to 0. This indicates that the first two subsets should use texture[0] and material[0] in the original attribute array.

With this in mind, when rendering a skin in managed mode, our rendering strategy will need to be changed a little. We will still loop through each subset and render it; the only different now is what we consider to be the number of subsets and what we use to index into the attribute array. Previously, we knew that if a managed mesh had five attributes, then it has five subsets, and we could simply render the mesh using the following pseudo-code strategy:

```
for ( i = 0 ; i < NumSubsets ; i++)
{
     pMesh->DrawSubset[i];
}
```

In the above code, we are looking at how the actor might render the various subsets of a CTriMesh stored in a mesh container. If we remind ourselves of the code inside CTriMesh::DrawSubset we can see the rendering logic is simple:

Excerpt from CObject.cpp in previous lab projects

```
void CTriMesh::DrawSubset( ULONG AttributeID )
{
    LPDIRECT3DDEVICE9 pD3DDevice = NULL;
    LPD3DXBASEMESH    pMesh      = m_pMesh;
```

```
    // Retrieve mesh pointer
    if ( !pMesh ) pMesh = m_pPMesh;
    if ( !pMesh ) return;

    // Set the attribute data
    if ( m_pAttribData && AttributeID < m_nAttribCount )
    {
        // Retrieve the Direct3D device
        pD3DDevice = GetDevice( );

        pD3DDevice->SetMaterial( &m_pAttribData[AttributeID].Material );
        pD3DDevice->SetTexture( 0, m_pAttribData[AttributeID].Texture );

        // Release the device
        pD3DDevice->Release();

    } // End if attribute data is managed

    // Otherwise simply render the subset(s)
    pMesh->DrawSubset( AttributeID );

}
```

Take a while to examine the above function and see if you can detect the problem this function will cause going forward. This function implies the 1:1 mapping between the subset ID which is about to be rendered and the texture and material stored in the attribute array. The ‘if’ statement basically determines whether the mesh’s attribute array exists. If it does, this then is a managed mesh and this table contains a texture and material for each subset. As you can see, the subset ID passed into the function is used to fetch the texture and material from the attribute array before making the call to the ID3DXMesh::DrawSubset function to render the triangles. If the attribute array does not exist, then this means this is a non-managed mode mesh and the scene will have already taken care of setting the correct texture and material for this subset. In this case, we simply render the subset.

We have been using this strategy for rendering managed mode meshes all along, and will continue to use the same strategy for managed regular meshes in our actor’s hierarchy. However, if the mesh we intend to render is a skin, then its mesh container will contain an array of D3DXBONECOMBINATION structures, one for each subset in the new mesh. The pseudo-code to render it inside the actor would be as follows (non-indexed example). In this example, pMesh is a pointer to a CTriMesh in our hierarchy.

```
for ( i = 0; i < NumSubsets; i++ )
{
    long AttribID = pBoneCombinationArray[i].AttribId;

    pMesh->DrawSubset[ i , AttribId ];
}
```

Of course, this is a pretty vague example but we will get to the actual code soon enough. The important point here is simply that the before rendering each subset in the skin, we fetch the original ID of the subset we are about to render from the bone combination structure. This is also the index of the texture and material that the original subset used, so we can use it to access the correct texture and material in our array before rendering the subset.

In order to facilitate this change, we will slightly modify our CTriMesh::DrawSubset method. Previously, this method had a single parameter, an attribute/subset ID. As we have seen, this is used to fetch a texture and a material from the mesh's array and set it on the device prior to rendering the requested subset. However, in the above example, we can see that when the actor is rendering a skinned mesh, it needs to pass two parameters into the CTriMesh::DrawSubset. The first parameter should be the attribute/subset ID as always. This is the subset in the underlying ID3DXMesh whose triangles we wish to render. However, now the subset IDs no longer match the position of their respective textures and materials in the managed mesh's attribute array. So, as the second parameter, we pass in an attribute override. In other words, the first parameter describes the physical subset in the mesh we wish to render, while the second parameter contains the index of the texture and material in the managed mesh's attribute array. Therefore, what the above code is doing, is looping though each subset in the new skinned mesh, and using its original subset ID (from the non-skinned mesh) to fetch the texture and material that should be used to render the subset.

Let us have a look at the new version of the CTriMesh::DrawSubset method and how this additional parameter is used. This additional parameter has a default value of –1, meaning no material override will be used and the 1:1 mapping between the subset ID and the attribute table is maintained as before. This means the actor can render managed mode regular meshes in exactly the same way as our prior lab projects simply by not passing in the second parameter.

The new CTriMesh::DrawSubset Method

```
void CTriMesh::DrawSubset( ULONG AttributeID, long MaterialOverride /* = -1 */ )
{
    LPDIRECT3DDEVICE9 pD3DDevice = NULL;
    LPD3DXBASEMESH    pMesh      = m_pMesh;
    ULONG             MaterialID = (MaterialOverride < 0) ?
                                    AttributeID : (ULONG)MaterialOverride;

    // Retrieve mesh pointer
    if ( !pMesh ) pMesh = m_pPMesh;
    if ( !pMesh ) return;

    // Set the attribute data
    if ( m_pAttribData && AttributeID < m_nAttribCount )
    {
        // Retrieve the Direct3D device
        pD3DDevice = GetDevice( );

        pD3DDevice->SetMaterial( &m_pAttribData[MaterialID].Material  );
        pD3DDevice->SetTexture ( 0, m_pAttribData[MaterialID].Texture );

        // Release the device
        pD3DDevice->Release();

    } // End if attribute data is managed

    // Otherwise simply render the subset(s)
    pMesh->DrawSubset( AttributeID );
}
```

As you can see, the new CTriMesh::DrawSubset code is almost identical except for the following line:

```
ULONG MaterialID = (MaterialOverride < 0) ? AttributeID : (ULONG)MaterialOverride;
```

What is happening here is quite simple. We are simply saying that if the material override is –1, then there is no material override and we should assume a 1:1 mapping between the subset ID and the element in the attribute array which contains the subsets texture and material (managed mode only). However, if a material override has been passed in (which will always be the case for a skinned mesh) then we are stating that the material override value should be used to index into the attribute array to fetch the correct texture and material for the subset being rendered.

If we have the actor in non-managed mode, things are quite different for skinned meshes. With regular meshes, we simply passed the loaded texture and material to an attribute callback function. This function (implemented by the CScene class in our code) would store the texture and material in some external array and would return the index of the element in this array back to the mesh. The mesh would then lock the attribute buffer of the mesh and would re-map all faces in the current subset being processed to this new ID value. The non-managed mesh did not store the textures or materials and actually had no idea what the global ID assigned to its triangles even meant. It doesn't matter though because the scene will handle setting the correct textures and materials before rendering those subsets, not the mesh itself. It was possible (as we have previously discussed) that after the global re-mapping of a non-managed mesh, its subset IDs might no longer even be zero based. For example, a mesh which was loaded from an X file with three subsets (0, 1, 2) might be remapped by the scene such that the subset IDs end up being (75, 104, 139). These are just arbitrary numbers as far as the mesh is concerned, but to the scene object, these are global IDs into its own texture and material arrays. In this example, the scene would know that the first subset would need to be rendered using texture[75] in its scene global texture array. There may be other non-managed meshes in the scene which also have subsets with an ID of 75. The scene would know that all of these subsets could be batch rendered because they all share the same texture. In other words, the re-mapping makes the subset IDs of a mesh global instead of local, allowing shared subsets across mesh boundaries.

While none of this is new to us, when an actor is in non-managed mode, we must handle skinned meshes very differently from regular meshes (which will still use this same strategy). We will still have the non-managed mode actor working exactly the same from the applications perspective and this will still allow us to re-map the skinned meshes subsets to use a global pool of textures and materials stored at the scene level. However, unlike the case of a regular mesh, we definitely must *not* lock the skinned mesh's attribute buffer and change the subset IDs of its triangles. Remember, the zero based subset IDs in the skinned mesh have a 1:1 mapping with the D3DXBONECOMBINATION array (calculated when the conversion into a skinned mesh happens), so if we go blindly fiddling with the attribute IDs of triangles in a skinned mesh, we will lose the ability to determine which D3DXBONECOMBINATION structure in its array is applicable to each subset in the skin. As a result, we will no longer know which bone matrices must be bound to the device prior to rendering eaxch subset. No, we must leave the actual subset IDs alone in the skinned mesh case so that we know that BoneCombination[5] for example, describes the bone matrices that must be set before we render subset[5] in the skinned mesh.

Of course, all is not lost. In fact, when you think about what we are trying to achieve you find that in many ways handling global subset IDs in the skinned case is actually easier than in the regular case. The whole point of this mode is really just to allow the mesh to use textures and materials stored in the

scene's database. We can do that simply by re-mapping the AttribId member of each bone combination structure instead of having to re-map the attribute IDs of the triangles themselves.

As a quick example, let us assume that our actor is in non-managed mode. We know that this will mean that that an application defined attribute callback function will have been registered with the actor and used to re-map all the attribute IDs into global IDs. The following snippet of code shows how after a mesh is loaded, if it is in non-managed mode, we loop through each subset in the mesh and call the attribute callback function. This function is called for each subset and passed the texture and material which the scene can store somewhere. The function will return a new global ID for that subset. This is a snippet from the CreateMeshContainer method we have been using in previous projects. In this code, you are reminded that we collect all the new global subset IDs in the local pAttributeRemap array. At the end of this code, each element in this array will describe the new global ID's of each subset.

```
for ( i = 0; i < NumMaterials; ++i )
{
   if ( Callback.pFunction )
   {
      COLLECTATTRIBUTEID CollectAttributeID = (COLLECTATTRIBUTEID)Callback.pFunction;
      AttribID = CollectAttributeID( Callback.pContext, pMaterials[i].pTextureFilename,
                                     &pMaterials[i].MatD3D, &pEffectInstances[i] );

       // Store this in our attribute remap table
       pAttribRemap[i] = AttribID;

       // Determine if any changes are required so far
       if ( AttribID != i ) RemapAttribs = true;

    } // End if callback available

} // Next Material
```

In our previous projects, once this re-map array has been compiled, our next task is to lock the attribute buffer of the mesh and re-map the IDs of each triangle in the mesh so that they now use the global IDs.

Re-mapping the attributes of a regular mesh

```
ULONG * pAttributes = NULL;

// Lock the attribute buffer
pMesh->LockAttributeBuffer( 0, &pAttributes );

// Loop through all faces
for ( i = 0; i < pMesh->GetNumFaces(); ++i )
{
     // Retrieve the current attribute ID for this face
      AttribID = pAttributes[i];

      // Replace it with the remap value
      pAttributes[i] = pAttribRemap[AttribID];
} // Next Face

// Finish up
pMesh->UnlockAttributeBuffer( );
```

The above code is not new to us and shows what we have always done and will continue to do when working with regular meshes in non-managed mode. It locks the attribute buffer, steps through each face and changes its attribute from a local subset ID to a global subset ID using the re-mapping information stored in the pAttribRemap array compiled in the previous section of code. This code will still be employed for regular meshes, but as discussed previously, if we start fiddling about with the attribute IDs of a skinned mesh, we will loose the 1:1 mapping between subset IDs and elements in the bone combination table. What we do in the non-managed skinned case is the following:

```
LPD3DXBONECOMBINATION pBoneComb =
  (LPD3DXBONECOMBINATION)MeshContainer->pBoneCombination->GetBufferPointer();

for ( i = 0; i < pMeshContainer->AttribGroupCount; ++i )
{
   // Retrieve the current attribute / material ID for this bone combination
   AttribID = pBoneComb[i].AttribId;

   // Replace it with the remap value
   pBoneComb[i].AttribId = pAttribRemap[AttribID];
}
```

In the non-managed skinned case we first fetch a pointer to its bone combination array. Remember, this array will be generated when we use the ID3DXSkinInfo::ConvertToBlendedMesh function. In this example it is assumed that this bone combination table was stored in the mesh container.

We next loop through each subset in the mesh, which is equal to the number of D3DXBONECOMBINATION structures. Instead of re-mapping the actual triangle attribute, we simply re-map the original attribute ID stored in each D3DXBONECOMBINATION structure.

So, where does this leave us? If we have an actor in non-managed mode that contains a single skinned mesh, we will end up with a mesh that has no internally stored attribute information (just as in the regular non-managed mode case). Unlike the non-managed mode regular mesh case, its subset IDs will still be local and zero based. However, what the mesh container will contain is a bone combination structure for each subset in the new mesh. Each bone combination structure will contain (inside the AttribId member) the new global ID that the scene can use to identify the texture and material that should be used to render each subset.

This might sound like a huge shift in our scene's rendering strategy if it wishes to render an actor in managed mode. However, no changes need to be implemented by the scene. Below, you can see a slightly compacted version of the scene's rendering code. For simplicity, each object in the scene is assumed to be a CActor:

```
void CScene::Render( CCamera & Camera )
{
    ULONG i, j;
    long  MaterialIndex, TextureIndex;

    …

    // Process each object
```

```
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CActor    * pActor = m_pObject[i]->m_pActor;

        // Set up actor / object details
        pActor->SetWorldMatrix( &m_pObject[i]->m_mtxWorld, true );


        // Loop through each scene owned attribute
        for ( j = 0; j < m_nAttribCount; j++ )
        {
           // Retrieve indices
           MaterialIndex = m_pAttribCombo[j].MaterialIndex;
           TextureIndex  = m_pAttribCombo[j].TextureIndex;

           // Set the states

           m_pD3DDevice->SetMaterial( &m_pMaterialList[ MaterialIndex ] );
           m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ]->Texture );

           // Render all faces with this attribute ID
          pActor->DrawActorSubset( j );

        } // Next Attribute

    } // Next Object

}
```

This rendering code assumes that all the actors in the scene have been created as non-managed actors. As such, the scene contains global arrays of textures and materials used by all meshes in all actors in the scene. As you can see, our rendering strategy has not changed. We simply loop through each actor and set its world matrix to position it correctly in the scene. For each actor, we loop through every global attribute combination managed by the scene and set the corresponding texture and material for that global attribute combination. Finally, we pass the index of that global attribute ID into CActor::DrawActorSubset where we expect the matching triangles to be rendered with this single call.

When we had only regular meshes in the actor's hierarchy, the CActor::DrawActorSubset method was quite straightforward. It would simply traverse the hierarchy looking for mesh containers. Once a mesh container had been located the global ID passed into the DrawActorSubset method could be directly passed to the ID3DXMesh::DrawSubset method to render the corresponding triangles. This is because triangle attributes IDs would have been mapped to these global IDs. However, with non-managed skins a slightly different rendering approach will be needed since the link between the subset we wish to render and the global attribute ID is in the AttribId member of each element in the D3DXBONECOMBINATION array.

When we wish to draw the subset of a non-managed skin, we will loop through each of its bone combination structures and find the subset that uses this global ID (if it exists). Below we see a very simplified rendering example which will be fleshed out later when we cover the actual source code. It demonstrates how the actor would render the requested subset for a skinned mesh. AttributeID is assumed to contain the subset ID passed in by the scene describing the global subset it wishes to render.

```
LPD3DXBONECOMBINATION pBoneComb =
      (LPD3DXBONECOMBINATION)MeshContainer->pBoneCombination->GetBufferPointer();

for ( i = 0; i < pMeshContainer->AttribGroupCount; ++i )
{
     if (pBoneComb[i].AttribId != (ULONG)AttributeID )
          continue;

     // Set up the bone stuff here ( shown later )

     // Draw CTriMesh
     pMesh->DrawSubset( i , pBoneComb[i].AttribId );
}
```

In this example we are showing special case code to render the subset of a skinned mesh. What is important about this code is that it works for both managed and non-managed modes. To understand why, let us step through what is happening.

We first fetch the bone combination array and loop through each element (subset) stored there. For example, let us imagine that we wish to render global attribute ID 57. We loop through each bone combination structure and test to see if the current subset being processed has 57 in the AttribId member of its bone combination structure. If not, then we skip the current subset and continue the search. Once we do find a subset that has a 57 match, we call the CTriMesh::DrawSubset method which we looked at earlier in this lesson. This function is the means by which this all works out nicely. As the first parameter we pass in the subset we wish to render and as the second parameter we pass in the AttribId member as the material override. If this is a managed mode mesh, then the material override is used to select the correct texture and material in its internal attribute array and send it to the device prior to rendering the subset. If this is a non-managed mode mesh then the material override will be ignored. For example, if we find that subset 4 in this mesh has 57 in the AttribId member of its bone combination structure, this simply means that we need to render subset 4 of the mesh (the material and texture have already been set by the scene). This will all come together very quickly when we look at the new rendering and loading code for CActor later in the workbook.

Summing up, the application's interaction with the actor will not change from earlier projects, even when skinned meshes exist in the hierarchy. The actor loading and rendering code will take care of rendering the special cases. It will use different rendering and loading code if a mesh is a regular mesh or a skinned mesh as well as special case code for managed and non-managed flavors.

Non-Indexed Skinned Mesh Rendering Optimization

As discussed in the textbook, we will sometimes be confronted with a mesh that has some subsets that can be rendered in hardware and others that can only be rendered in software. In our rendering code, we will have to render these meshes using a two pass approach. In the first pass we could disable software vertex processing and loop through each subset in the mesh. For each subset, we would test how many bones influence it, and if it is smaller or equal to the number supported by the hardware, we would render the subset. After we have looped through every subset, we will have rendered all subsets in the mesh that were hardware compliant using hardware acceleration. In the second pass, we would enable

software vertex processing and do a search for subsets that need to be rendered in software, Once again, we would loop through every subset in the mesh, but this time we would count the bone influences of each subset and only render subsets where the number of bone influences is larger than the number supported by the hardware. By the end of this second pass, all subsets will have been rendered.

While this approach is certainly more efficient then just rendering all subsets with software vertex processing, it does require doing two conditional passes through each subset of the mesh. Furthermore, in each pass, for each subset, we need to count the number of bones that influence that subset. This involves counting the number of Bone IDs in the subset's corresponding bone combination structure. All of this looping, counting, and testing is less than ideal for a time critical function, so we will implement a separate re-map buffer in the non-indexed case that contains the subset IDs batched by hardware compliance. For example, imagine that we have a non-indexed skin with ten subsets. Also imagine that instead of performing the tests mentioned above, we perform them at startup when the mesh is first created. Let us assume that in this example, we learn that subsets 2, 4, 5 and 8 have more bone influences than the hardware can handle.

Without optimization, our render logic would look something like this:

```
// First Pass ( Hardware Subset Pass )
pDevice->SetSoftwareVertexProcessing( false );

For ( i = 0; i < NumAttributes; i++)
{
    If ( Subset[i] == Hardware Compliant ) Render Subset[i];
}

// Second Pass ( Software Subset Pass )
pDevice->SetSoftwareVertexProcessing ( true );

For ( i = 0; i < NumAttributes; i++)
{
    If ( Subset[i] != Hardware Compliant ) Render Subset[i];
}
```

This approach is less than optimal. We can perform these tests when the mesh is first created so we should be able to store the subset IDs in an additional array grouped by hardware/software compliance. In fact, that is exactly what we will do. When the mesh is first created, we will loop through the subsets in two passes. In the first pass we will search for all subsets that are supported in hardware and add them to this additional re-map array. In the second pass we will search for all subsets that are not supported by hardware and add them to the array as well.

Using our current example, if a mesh has ten subsets, we will need a re-map array that contains ten subset IDs. As mentioned, subsets 2, 4, 5 and 8 cannot be rendered using hardware. Therefore, in the first pass, we would add subsets IDs 0, 1, 3, 6, 7, and 9 to the re-map array, and in the second pass subset IDs 2, 4, 5 and 8 would be added. The sorted array of subset IDs would now look like this:

[0, 1, 3, 6, 7, 9, 2, 4, 5, 8]

All we are doing is using this array to store the subset IDs in an optimal rendering order. Additionally, our mesh container can store the index into this array where the software subsets start. In that case, our rendering logic for a non-indexed mesh would now look like the following.

```
// First Pass ( Hardware Subset Pass )
pDevice->SetSoftwareVertexProcessing( false );

For ( i = 0; i < SoftwareStartIndex; i++)
{
    Render Subset[ Remap[I] ];
}

// Second Pass ( Software Subset Pass )
pDevice->SetSoftwareVertexProcessing ( true );

For ( i = SoftwareStartIndex; i < NumAttributes; i++)
{
    Render Subset[ Remap[i] ];
}
```

Although it looks like we do two passes here, we only ever have to loop through the entire subset array once. In the above code, it is assumed that SoftwareStartIndex was calculated at mesh creation time and contains the index into the re-map array where the software subsets begin. Remap is assumed to be the array that contains the ordered subset IDs. We thus loop through each subset but use the re-map array to fetch the ID of the subset that must be rendered next.

We have now discussed many of the hurdles we must overcome to implement skinned mesh support in CActor. While some of this information may be a bit vague or seem very complicated at the moment, our detailed walkthrough of the source code should give you all the insight you need.

Lab Project 11.1 - Implementation Goals

1) Extend CActor to provide the loading and managing of hierarchies that contain skinned meshes.

2) Allow CActor to handle the loading and managing of hierarchies that contain mixed geometry types. The actor should be able to render itself even if the hierarchy contains multiple static and skinned meshes.

3) Encapsulate skinned mesh support inside CActor such that providing skinned support does not place any additional burden on the application. The application should be able to simply load the X file using CActor::LoadActorFromX like it always has and have the details of loading and rendering the different geometry types handled behind the scenes.

4) Implement our skinned support such that managed and non-managed actor modes are still available to the application.

Lab Project 11.1 - The Source Code

Most of the changes to the CActor class will be added to existing functions. The primary changes will show up in functions which load and render the actor. These changes are focused around the CAllocateHierarchy::CreateMeshContainer method which is where the code that takes the loaded mesh and coverts it into a skin is located. CActor::DrawSubset and CActor::DrawMeshContainer are also going to be enhanced to facilitate multiple rendering techniques for regular meshes and skinned meshes of various flavors (software, indexed, and non-indexed). A few new helper methods will also be added to CActor to assist in the building process.

Our New Mesh Container

Before we look at the CActor class declaration, we first need to extend our derived D3DXMESHCONTAINER structure. As you will recall, each frame that has a mesh attached will have that mesh stored in a mesh container. Previously this mesh container has had to hold nothing more than a pointer to a CTriMesh, but now we have much more per-mesh information to worry about. For example, a skinned mesh will need to have access to the bone combination table so that it knows which matrices to use when transforming each of its subsets. It will also need to store two arrays of matrix information. As discussed in the textbook, in order to render a skinned mesh, we will need access to each of its bone matrices. The bone matrices are the absolute frame matrices in the hierarchy which will be updated every time an animation is applied to the bones. We do not want to have to traverse the hierarchy each time we need to collect the bones before we set them on the device, so we will traverse the hierarchy at load time and store matrix pointers to each bone in the mesh container. We will also store (in a separate array) the bone offset matrices for each bone used by the mesh. There will be a 1:1 mapping between the bone matrix pointer array and the bone offset matrix array such that, if a subset we are about to render uses bone [5], we will multiply bone[5] with bone offset[5] and set the resulting matrix on the device. There are many more members we will need to add, so let us have a look at our new D3DXMESHCONTAINER derived structure.

```
struct D3DXMESHCONTAINER_DERIVED : public D3DXMESHCONTAINER
{
    CTriMesh   * pMesh;
    bool         Invalidated;
    ULONG        SkinMethod;
    D3DXMATRIX * pBoneOffset;
    D3DXMATRIX** ppBoneMatrices;
    DWORD        AttribGroupCount;
    DWORD        InfluenceCount;
    LPD3DXBUFFER pBoneCombination;
    ULONG        PaletteEntryCount;
    bool         SoftwareVP;
    ULONG      * pVPRemap;
    ULONG        SWRemapBegin;
    LPD3DXMESH   pSWMesh;
};
```

Let us discuss the members of this now quite large structure. Remember, the members you see above are in addition to the members inherited from D3DXMESHCONTAINER. Every mesh in our hierarchy will be contained in one of these structures and attached to a parent frame. If the mesh container contains a regular mesh, most of these members will be unused.

CTriMesh *pMesh (All Meshes)
We have had this member in all previous versions of CActor. It stores the CTriMesh pointer which points to the mesh that will be rendered when this mesh container is encountered during the rendering pass of the hierarchy. This mesh may be in managed or non-managed mode and may contain a regular mesh or a skin. The only difference between a regular mesh and a skin is that the skin will contain vertex weights and the rest of the members in this structure will be used to determine which matrices should be set on the device prior to rendering it. However, what this mesh represents in the various different skinning modes needs some explanation.

Regular Mesh
If the mesh container stores a regular mesh then this is where that regular mesh is stored. It is this mesh that is rendered when the mesh container is encountered during the rendering pass of the hierarchy. It represents the model space mesh data that was loaded from the X file.

Non-Indexed & Indexed Skinning Mesh
If the mesh container stores a mesh that will be used as a skin by the pipeline, this CTriMesh will contain the skin in its reference pose. This is the mesh that was generated with a call to either the ConvertToBlendedMesh or ConvertToIndexedBlendedMesh functions. This mesh will be multi-matrix transformed by the pipeline during the rendering pass of the hierarchy. It does not contain the exact data loaded from the X file; instead it contains the original mesh converted into a skinned mesh. The vertices will contain weights, but the geometry will still represent the reference pose originally loaded from the X file.

Software Skinned Mesh
The use of the CTriMesh in the software skinned case may be slightly confusing at first. You will recall from the textbook that when performing software skinning we actually require two ID3DXMesh's (or at least two vertex buffers). We have a source mesh which contains the original model space mesh geometry loaded from the X file (the reference pose) and we also need a destination mesh to store the transformed vertices generated by the ID3DXSkinInfo::UpdateSkinnedMesh method. This needs to be done whenever bones in the hierarchy have been animated that would cause the shape or position of the skin to change.

Therefore, the source mesh will always contain the untransformed skin that was loaded, and the destination mesh is continually updated when transformations are applied. The destination mesh is always the mesh that gets rendered since it contains the world space version of the skin. When this mesh is rendered, the device world matrix should be set to identity.

You will see in a moment that our new extended mesh container structure stores an additional pointer to an ID3DXMesh interface called pSWMesh. This means the mesh container has the ability to store two ID3DXMesh interfaces (the other is inside the CTriMesh). We can use one mesh to store the original reference pose vertex data and another to transform into and render

from. So, in the software skinning case does the CTriMesh member contain the source mesh or the destination mesh?

The obvious choice would be to use the CTriMesh as the destination mesh since this is the mesh that we render each time we traverse the hierarchy. We could store the model space reference pose geometry in the other ID3DXMesh (pSWMesh) and transform it into the CTriMesh's vertex buffer whenever the hierarchy frames are updated. After all, in managed mode the CTriMesh object contains all the attributes it needs to render itself, so the CTriMesh should contain the transformed data and be used as the desination mesh. However, this is not really consistent with how all other mesh types that we can store in the hierarchy store their data. This makes such a design choice awkward for a number of reasons.

For starters, in both the regular and pipeline skinned mesh cases, the CTriMesh always stores the untransformed mesh (it is transformed in the pipeline). Thus, if the application wanted to explore the hierarchy and access the data of any mesh contained therein, it would expect the mesh data of the CTriMesh to be defined in model space. We would break with this approach if we stored the transformed vertex data in this object in the software skinned case only. But again, this mesh is also where the attribute data is stored and it is the mesh that we would want to use for rendering purposes. This seems to create a small dilemma.

As it happens, we will perform a little trick when rendering a software skin which will allow us to have the best of both worlds. You will recall that the CTriMesh class has an Attach method which allows us to attach and detach any ID3DXMesh as its underlying mesh data. With this in mind we will do the following:

The CTriMesh object will always house the ID3DXMesh object that contains the model space geometry in its reference pose and the additional ID3DXMesh (pSWMesh) will be the mesh that we transform the mesh data into each time the actor's hierarchy is updated. This allows us to stay consistent with how the actor works in all other modes. That is, if an application wished to retrieve the vertex buffer of any mesh in the hierarchy, it will always gain access to the model space vertex data. In the software case, the transformed mesh data will be stored in the additional ID3DXMesh pointed to by the pSWMesh variable. All is well so far, except for the fact that the standalone ID3DXMesh (pSWMesh) that contains the transformed vertices does not have any information about which subsets use which textures and materials. It also does not have the useful rendering methods that automate the setting of textures and materials in managed mode since it is not encapsulated in a CTriMesh. It is CTriMesh that provides us with these tools.

Every time the hierarchy is updated we will need to rebuild any software skins. We will use the ID3DXSkinInfo::UpdateSkinnedMesh method to transform the model space vertices from the CTriMesh vertex buffer into our destination ID3DXMesh (pSWMesh) vertex buffer. However, when it comes time to render this mesh, we will temporarily detach the CTriMesh's underlying ID3DXMesh and in its place we will attach our transformed mesh (pSWMesh). We can then use the CTriMesh methods to render the mesh with full access to all texture and materials. Once we have rendered it, we will detach the transformed mesh and restore the CTriMesh's original ID3DXMesh that contains the untransformed reference pose data. This approach addresses all or our concerns.

bool Invalidated (Software Skinning Only)
This boolean member variable is applicable only to software skinning. In the textbook we learned that when performing software skinning, we use the ID3DXSkinInfo::UpdateSkinnedMesh method to multi-matrix transform vertices from a source mesh into a destination mesh. The pipeline plays no part in the world space transformation process since we essentially hand a world space destination mesh to the pipeline for rendering. However, the problem is that if the bones in the hierarchy are updated by an animation for example, the destination mesh will need to be rebuilt. That is, we will have to transform the vertices from the source mesh into the destination mesh all over again so that the bone position changes are reflected in the new destination mesh. Of course, we do not wish to do this every time we render the mesh since this would hurt performance if we did so needlessly. We only want to do it if the hierarchy has been updated. Therefore, when an animation is applied to the hierarchy, the mesh container will become invalid. During our UpdateFrames method, for each frame we visit during the update, we will set the Invalidated member of any attached mesh container to true.

Before we render a mesh container, we will first test to see if it contains a software skin, and if so, check its validity status. If it invalid, we rebuild the destination mesh (using UpdateSkinnedMesh), set the Invalidated boolean back to false, and then render the mesh. Moving forward, when we render the actor, the destination mesh containing the world space skin will simply be rendered without being rebuilt. It will only have to be rebuilt from the source mesh data when another animation is applied to the hierarchy which invalidates the mesh container all over again. This is a handy flag since it removes any management burden from the application. The application can simply apply animations and render the actor and the actor will automatically rebuild invalidated software skins before they are rendered.

ULONG SkinMethod (All Skinned Meshes)
This member contains the skinning method being used by the mesh stored in this container. This is set using the CActor::SetSkinningMethod function prior to the hierarchy being constructed. The default mode for the actor is SKINMETHOD_AUTODETECT. However, depending on the hardware support available, the actor may create a hierarchy that contains skins created for different skinning techniques. For example, it may find that some can be created as hardware skins and others will have to be created as software skins. So this member may not be the same across all mesh containers in the hierarchy.

```
enum SKINMETHOD    { SKINMETHOD_INDEXED = 1,
                     SKINMETHOD_NONINDEXED = 2,
                     SKINMETHOD_SOFTWARE = 3,
                     SKINMETHOD_AUTODETECT = 4,
                     SKINMETHOD_PREFER_HW_NONINDEXED = 8,
                     SKINMETHOD_PREFER_SW_INDEXED = 16 };
```

This member is not used for regular meshes.

D3DXMATRIX * pBoneOffset (All Skinned Meshes)
This array is allocated to contain the bone offset matrices for each bone matrix in the hierarchy. It is used by all skinned meshes and will be populated at mesh creation time. When D3DX is loading the hierarchy and a skin is encountered in the X file, it will call our ID3DXAllocateHierarchy::CreateMeshContainer callback function passing an ID3DXSkinInfo interface pointer. The ID3DXSkinInfo object contains all the skinning information, such as which weights apply to which vertices for each matrix in the hierarchy. For each bone, it will also store a bone offset matrix.

We will extract those bone offset matrices and store them in this array so that we can access them during the rendering pass. When a subset uses bone[5] for example, it means we need to combine bone offset matrix[5] with bone matrix[5] (stored in the array below) and set this combined matrix in the appropriate matrix slot on the device prior to rendering that subset.

D3DXMATRIX ppBoneMatrices (All Skinned Meshes)**
Every frame in the hierarchy that is attached to vertices in a skin is referred to as a bone. In fact, it will ultimately be the absolute frame matrix (not relative matrix) which is the final bone matrix. Before we set a bone matrix on the device, we must first combine it with its bone offset matrix. The problem is that our bone matrices are stored in the hierarchy and would require a hierarchy traversal every time we need them to render a subset. Instead, at mesh creation time, we will fetch all the bone offset matrices from the ID3DXSkinInfo object and store them in the array described above. For each bone offset matrix, the ID3DXSkinInfo object will also contain the name of the bone (the frame name) in the hierarchy to which it corresponds. Therefore, in a one time pass, we will traverse the hierarchy and store pointers to all the bone matrices (absolute frame matrices) in this matrix pointer array so we will have direct access to them during rendering. By storing matrix pointers it also means this array will not have to be rebuilt when the matrices have animations applied to them. What we will have is a 1:1 mapping between the bone offset array and this bone matrix pointer array. This will allow us to efficiently fetch a bone matrix and its corresponding bone offset matrix before combining them and sending the result to the device.

DWORD AttribGroupCount (Pipeline Skinned Meshes Only)
This member will contain the number of subsets in the skinned mesh. This may be quite different from the number of subsets that were described in the original mesh that was passed into the CreateMeshContainer callback. Remember, the ID3DXSkinInfo::ConvertToBlendedMesh function will convert the original mesh into a skinned mesh and, when doing so, it may need to break the mesh into new subsets based on bone contribution. This member contains the final number of subsets in the mesh, which corresponds to the number of D3DXBONECOMBINATION structures in the bone combination array (described next).

LPD3DXBUFFER pBoneCombination (Pipeline Skinned Meshes Only)
When one of the pipeline skinning methods is being used by our actor, we will need to convert the original ID3DXMesh into a skinned mesh using the ID3DXSkinInfo::ConvertToBlendedMesh or ID3DXSkinInfo::ConvertToIndexedBlendedMesh functions. As we have discussed in the accompanying textbook, this function will create a new mesh with the correct number of weights allocated in the vertices. The resulting mesh will also include the weights and matrices used by each vertex, subdividing subsets where necessary.

These skinned mesh conversion functions will return an ID3DXBUFFER which contains an array of D3DXBONECOMBINATION structures. There will be one element in this array for each subset in the newly created skinned mesh. We will store the buffer in this member variable of the mesh container. We will need access to the bone combination buffer when rendering pipeline skinned meshes because each structure in the array tells us the original subset ID for each subset. We use this to index into a managed mode mesh's attribute table to fetch the texture and material for that subset. Each bone combination structure in this array also informs us of the bone matrices that need to be sent to the device for each subset. These bone indexes are stored in the BoneId array inside the D3DXBONECOMBINATION structure. For example, BoneId[2]=57 informs our rendering code that before rendering this subset, we

must fetch bone matrix[57] from the bone matrix array, bone offset matrix[57] from the bone offset array and combine them together before assigning them to matrix slot[2] on the device. For non-indexed skinned meshes, the BoneId array of each subset will, at most, contain four valid elements, as only a maximum of four matrices can ever be set on the device. For indexed skinning, the BoneId array could contain up to 256 valid elements since we have the ability (hardware permitting) to set 256 bone matrices on the device simultaneously. This member is not used for software skinning or for regular meshes.

DWORD InfluenceCount (Pipeline Skinned Meshes Only)

This member is only used by pipeline skinned meshes. It helps us set the correct D3DRS_VERTEXBLEND render state prior to rendering a given subset. The value of this member will be returned to us by the ConvertToBlendedMesh and ConvertToIndexedBlended mesh functions and has a slightly different meaning in each case:

Non-Indexed Skinning

In the non-indexed case, it informs us of the size of the BoneId array in each D3DXBONECOMBINATION structure. For example, if this value was 4, we would know that the BoneId array for every subset will have 4 elements. Not all elements will necessarily be used, as a given subset may only use 1 or 2 bones for example. In such cases, the unsused elements will be set to UINT_MAX. We need to know how large the BoneID arrays are because our code will need to search through a subset's BoneId array prior to rendering the subset to collect valid bone indices. If we did not know the size, we would not know how many elements in the BoneId array to test. The value returned from ID3DXSkinInfo::ConvertToBlendedMesh will be the maximum number of bones/matrices that influence a single subset in the mesh. This will assure us that the BoneId array will contain this many elements even though all subsets may not use them all. When rendering a non-indexed skinned mesh, we can use this value to test the BoneId array for valid bone indices and also count how many bones that actual subset is using. This allows us to set the D3DRS_VERTEXBLEND render state to the correct number of weights for each subset prior to rendering it. This allows us to rid ourselves of redundant zero weight contribution matrix multiplications.

Indexed Skinning

In the indexed case this value has a slightly different meaning although it is still used to set the D3DRS_VERTEXBLEND render state as in the non-indexed case. In order to render a subset using indexed skinning, we must correctly configure the D3DRS_VERTEXBLEND render state to correctly inform the pipeline of the number of weights in each vertex. Unlike the non-indexed case where we can adjust this render state on a per-subset basis to select only the weights actually used, in the indexed case it must be set such that it informs the pipeline of the number of total weight components our vertex has. This is because the pipeline will need to know where the Matrix Indices Component (the last weight) is stored in the vertex.

Therefore, unlike the non-indexed case, we cannot make the optimization of setting this render state on a per subset basis so that only non-zero weight matrices are multiplied. We must inform the pipeline of the exact layout of weights in the vertex structure. Since the ConvertToIndexedBlendedMesh function will create a vertex structure with enough weights to cater for the vertex with the most bone influences, every vertex will have the same number of weights and have their indices in the same position. Therefore, if this value tells us how we

should set the D3DRS_VERTEXBLEND render state, it also tells us the maximum number of bones that influence a single vertex in the mesh.

Unlike the non-indexed case, this value does not tell us of the size of the BoneId array contained in each subset's D3DXBONECOMBINATION structure. In indexed mode, the two concepts are decoupled. The number of weights in a vertex can still be 4 at most, but a single subset of triangles could index into a palette of, for example, 100 matrices. In this case then, the BoneId array for each subset would have to be large enough to hold 100 elements. So, when setting the bone matrices for an indexed subset, we would need to loop through 100 BoneId elements checking for valid bone matrix IDs. Obviously, not all subsets would use 100 matrices in this mesh, but if one subset did, the BoneId array for every subset would be large enough to hold this many elements, even if many elements were unused.

In the indexed skinning case, the size of the BoneId array is actually equal to the size of the device matrix palette that we wish to use. This is calculated in a separate step and stored in the PaletteEntryCount member of the mesh container (described next).

ULONG PaletteEntryCount (Pipeline Indexed Skinning Only)
This member will be used for indexed skinned meshes and contains the current size of the matrix palette being used on the device. This number also describes the number of elements in each subset's BoneId array (for indexed skins only) since the BoneId array must have one element for each matrix slot on the device that a vertex may index.

bool SoftwareVP (Pipeline Indexed Skinning Only)
This Boolean member is used for pipeline indexed skinning only. It is set during the mesh creation phase to inform our renderer that the device will not be able to transform this mesh in hardware and that we must enable software vertex processing before making the draw call. We will typically set this to true if we discover, during mesh creation, that there exists a face (or many faces) that have more bone influences than there are slots in the matrix palette. Unlike the non-indexed case where we may have some subsets that can be hardware transformed and some that cannot, in the indexed case, if there is a single triangle in that mesh that has more bone influences than the device has to offer, the entire mesh must be transformed using software vertex processing.

We determine whether an indexed skin can be hardware transformed and rendered by testing the maximum number of influences per face against the number of slots available in the matrix palette. For example, if there is a triangle that has 12 bone influences, but the device only support 8 matrix slots, this device cannot transform this mesh in hardware and software vertex processing must be used instead. As long as the device supports at least 12 matrices we will always be able to render the indexed skin in hardware. This is because a triangle has three vertices and each vertex can be influenced by four bones at most. As a result, we will never encounter a situation where a single triangle is influenced by more than twelve bones. As long as at least twelve matrices are available, the ConvertToIndexedBlendedMesh function will be able to divide the mesh into hardware compliant subsets (even if a given subset only has one triangle in it).

ULONG * pVPRemap (Pipeline Non-Indexed Skinning Only)
Earlier we learned that in the non-indexed case, we may have some subsets that can be rendered using hardware vertex processing and others that require software vertex processing. We found out that instead of having to perform two complete passes (hardware and software) through the all the subsets at render time, we could determine (at mesh creation time) which subsets belong to which pool and group the subset IDs in a new array. We could loop through all the subsets, find the hardware capable ones and add them to the array first, then loop through the subsets again, find all the non-hardware capable subsets and add them to this same array. At the end of this process, we would have all the hardware capable subsets stored at the beginning of the array followed by the software ones. When rendering the non-indexed skin, we can simply render all the hardware capable subsets stored at the beginning of the array first, then enable software vertex processing and render the rest of the subsets in the array. This provides a nice optimization in our rendering code beyond what we discussed in the textbook.

This member will point to an array of subset IDs in the mesh, ordered into two groups (hardware and software subsets). The following member of the mesh container will store the index into this array where the software subsets begin so that during rendering we know exactly when to enable software vertex processing. This will eliminate all hardware compliance tests that were originally performed per-subset in the rendering code examples we discussed in the textbook.

This member is used only by non-indexed skins, since we can never render an indexed skin using a mixture of hardware and software subsets.

ULONG SWRemapBegin (Pipeline Non-Indexed Skinning Only)
This member is used alongside the array described previously. It stores the index into the pVPRemap array where the software subset IDs begin. We know that all subsets in the array preceding this index can be rendered with software vertex processing disabled, while all subsets starting from this index (to the end of the array) must be rendered with software vertex processing enabled.

This member is used only by non-indexed skins, since we can never render an indexed skin using a mixture of hardware and software subsets.

LPD3DXMESH pSWMesh (Software Skinning Only)
When performing pure software skinning we will need to transform the model space mesh data into world space ourselves prior to sending the vertex data to the pipeline for rendering. We perfom this task using the ID3DXSkinInfo::UpdateSkinned mesh. This method takes an array of bone and bone offset matrices, a source mesh, and a destination mesh. The function uses the passed matrix information to multi-matrix transform the model space vertices in the source mesh into world space vertices in the destination mesh. The destination mesh then contains the geometry that we pass to the pipeline. Therefore, unlike other skinning techniques, where transformation is performed in the pipeline, when performing software skinning we need an additional mesh (or at least an additional vertex buffer) to receive the results of the world space transformation.

This member (pSWMesh) is the destination mesh when software skinning is being used. Everytime the hierarchy is updated and the bones of the mesh have changed, we rebuild the world space version of the mesh using the ID3DXSkinInfo::UpdateSkinnedMesh method. The model space geometry is stored in our CTriMesh's underlying ID3DXMesh. It is this destination mesh that we render.

You will note that this is a regular ID3DXMesh and not a CTriMesh, so it would seem that we lose access to the handy rendering functions and texture and material information for each subset normally stored in the CTriMesh class. However, as discussed earlier, during rendering we will temporarily detach the CTriMesh's current ID3DXMesh and attach this destination mesh containing the world space version of the geometry. We will then render the CTriMesh normally which will use the world space geometry. After the mesh has been rendered, we will detach the destination mesh and re-attach the CTriMesh's original model space ID3DXMesh.

We have now covered the new additions to our mesh container structure. As you can see, it has now become a lot more complex with many more members. Also remember that the base class from which our mesh container is derived also has an ID3DXSkinInfo pointer member which we have not used in previous lab projects. This will now be used to store the ID3DXSkinInfo interface pointer for a mesh that is passed to the CreateMeshContainer method during the X file loading process. We will need this interface in many places, especially in the software skinning case where this interface actually contains the UpdateSkinnedMesh method that performs the software transformation of vertices from the model space source mesh into the world space destination mesh.

Source Code Walkthrough - CAllocateHierarchy

In previous lab projects, we have used CAllocateHierarchy (derived from ID3DXAllocateHierarchy) as our callback class. An instance of this class was passed to D3DXLoadMeshHierarchyFromX by our actor and its methods were used for frame and mesh container creation during hierarchy construction. You will hopefully remember that this class should implement the four functions of the base class: CreateFrame, CreateMeshContainer, DestroyFrame and DestroyMeshContainer.

You will also recall that every time a frame is encountered in the X file during the loading process, D3DX will call our CAllocateHierarchy::CreateFrame method. This will allow us to allocate the frame memory in whatever way best suits our application. We can then hand it back to D3DX for hierarchy attachment. This allows us to allocate frame structures derived from D3DXFRAME where we can add additional member variables and tailor it to our applications needs. For example, right from the beginning of our hierarchy usage, we used a derived frame structure that had an additional matrix that would be used to store the world space transform of the frame/bone. Likewise, whenever a mesh is encountered by D3DX during the rendering process, the CAllocateHierarchy::CreateMeshContainer function would be called and passed all the mesh information from the X file (e.g., the mesh geometry, the textures and materials, and any skinning information).

Our CAllocateHierarchy::CreateFrame function is unchanged in this lab project since we have no need for new members in our frame structure. However, the CAllocateHierarchy::CreateMeshContainer function will change significantly since we have to add code that creates managed and non-managed flavors of indexed skinned meshes, non-indexed skinned, meshes and software skinned meshes (in addition to the managed/non-managed standard meshes from before).

CAllocateHierarchy::CreateMeshContainer

We will cover the CreateMeshContainer function before we cover the code to CActor::LoadActorFromX. Our reason for doing this is because one of the first things the LoadActorFromX function does is called D3DXLoadMeshHierarchyFromX which then calls the CAllocateHierarchy::CreateMeshContainer method for every mesh contained in the X file. Therefore, with respect to program flow, it makes more sense to cover this function first since it is where all the mesh data is actually created, where all the bones are assigned to the skin, and where the mesh container's members are assigned their values. By the time the D3DXLoadMeshHierarchyFromX function returns program flow back to CActor::LoadActorFromX, any meshes (skins or regular) will have been created and attached to the frame hierarchy. There will be a tiny bit of additional work the actor will need to do after the hierarchy has been loaded to complete the setup process for skinned meshes, but we will get to that in a moment.

This function was already getting pretty large even before skinned support was added. Fortunately however, we have covered all of this code in detail previously and the first ¾ of the function code is unchanged from the prevous lab project. The new skinning support has been conveniently bolted onto the end of the function, for the most part. We will show the entire function in this section and discuss it. The first few parts of the function will only be briefly reviewed since we have already covered this code in previous lab projects. This function also makes calls into a few new CAllocateHierarchy helper functions, so we will cover the code to these functions immediately after.

D3DX passes this function the name of the mesh in the X file that is currently being loaded and the D3DXMESHDATA structure that contains the ID3DXMesh (which contains the geometry loaded from the file). It is also passed an array of D3DXMATERIAL structures describing the texture and material used by each subset in that mesh and an array of effect instances (one for each subset). We will ignore the effects array until Module III when we learn about rendering with effect files. The number of materials (subsets) used by the mesh and the mesh face adjacency information is also passed in. The penultimate parameter is one we have ignored in previous projects -- it is a pointer to an ID3DXSkinInfo interface which will be non-NULL when the mesh has skinning information defined in X file. Finally, we are passed the address of a D3DXMESHCONTAINER structure which, on function return, we will assign to point at the new mesh container that we are about to allocate and populate with the passed information.

```
HRESULT CAllocateHierarchy::CreateMeshContainer
                            ( LPCTSTR Name,
                              CONST D3DXMESHDATA * pMeshData,
                              CONSTD3DXMATERIAL*pMaterials,
                              CONST D3DXEFFECTINSTANCE *pEffectInstances,
                              DWORD NumMaterials,
                              CONST DWORD *pAdjacency,
                              LPD3DXSKININFO pSkinInfo,
                              LPD3DXMESHCONTAINER *ppNewMeshContainer)
{
    ULONG                      AttribID, i;
    HRESULT                    hRet;
    LPD3DXMESH                 pMesh          = NULL, pOrigMesh = NULL;
    D3DXMESHCONTAINER_DERIVED *pMeshContainer = NULL;
    LPDIRECT3DDEVICE9          pDevice        = NULL;
```

```
    CTriMesh                   *pNewMesh       = NULL;
    MESH_ATTRIB_DATA           *pAttribData    = NULL;
    ULONG                      *pAttribRemap   = NULL;
    bool                        ManageAttribs  = false;
    bool                        RemapAttribs   = false;
    CALLBACK_FUNC               Callback;

    // We only support standard meshes ( no progressive )
    if ( pMeshData->Type != D3DXMESHTYPE_MESH ) return E_FAIL;

    // Extract the standard mesh from the structure
    pMesh = pMeshData->pMesh;

    // Store the original mesh pointer for later use (in the skinning case)
    pOrigMesh = pMesh;

    // We require FVF compatible meshes only
    if ( pMesh->GetFVF() == 0 ) return E_FAIL;

    // Allocate a mesh container structure
    pMeshContainer = new D3DXMESHCONTAINER_DERIVED;
    if ( !pMeshContainer ) return E_OUTOFMEMORY;

    // Clear out the structure to begin with
    ZeroMemory( pMeshContainer, sizeof(D3DXMESHCONTAINER_DERIVED) );

    // Copy over the name
    // for the string belongs to the caller (D3DX)
    if ( Name ) pMeshContainer->Name = _tcsdup( Name );

    // Allocate a new CTriMesh
    pNewMesh = new CTriMesh;
    if ( !pNewMesh ) { hRet = E_OUTOFMEMORY; goto ErrorOut; }
```

The above section of code is essentially unchanged from previous lab projects. It first tests to see that it is not a progressive mesh that we have been passed (which we will not support in this lab project). If it is, we return immediately. We fetch the passed ID3DXMesh interface into a local pointer for ease of access and then test that this mesh has a valid FVF vertex format (once again, we exit if this is not the case). We will discuss non-FVF meshes in Module III. One minor change is the additional line that stores the original mesh pointer. We will use this pointer only when we are working with skins because the skinning process will change the mesh subsets and various D3DX utility functions will require that we have the original (pre-skinned) data available (for saving to file for example). Once these tests are out of the way, we allocate a new derived mesh container and initialize all its members to zero for safety. Then we copy the name of the mesh that was passed into the function into the ‘Name’ member of the mesh container. Finally, we allocate a new CTriMesh which will eventually be used to store our mesh data and attached to the mesh container.

Before attaching the passed ID3DXMesh to our new CTriMesh object, we test the FVF flags to see if the mesh contains normals. We require normals in our demo so that we can use the lighting pipeline. If no normals exist then we will clone the passed mesh into a new mesh that contains vertex normals and attach this cloned mesh to our CTriMesh object. If the passed mesh already contains normals then (as you can see in the ‘else’ code block), we simply attach it to our new CTriMesh.

```
    // If there are no normals add them to the mesh's FVF
    if ( !(pMesh->GetFVF() & D3DFVF_NORMAL) ||
          (pMesh->GetOptions() != m_pActor->GetOptions()) )
    {
        LPD3DXMESH pCloneMesh = NULL;

        // Retrieve the mesh's device (this adds a reference)
        pMesh->GetDevice( &pDevice );

        // Clone the mesh
        hRet = pMesh->CloneMeshFVF( m_pActor->GetOptions(),
                                    pMesh->GetFVF() | D3DFVF_NORMAL,
                                    pDevice, &pCloneMesh );
        if ( FAILED( hRet ) ) goto ErrorOut;

        //we don't release the old mesh here, because we don't own it
        pMesh = pCloneMesh;

        // Compute the normals for the new mesh
        if ( !(pMesh->GetFVF() & D3DFVF_NORMAL) )
            D3DXComputeNormals(pMesh, pAdjacency );

        // Release the device, we're done with it
        pDevice->Release();
        pDevice = NULL;

        // Attach our specified mesh to the new mesh
        pNewMesh->Attach( pCloneMesh );

        // We can release the cloned mesh interface here,
        // CTriMesh still has a reference to it so it wont be deleted
        pCloneMesh->Release();

    } // End if no vertex normal, or options are hosed.
    else
    {
        // Simply attach our specified mesh to the CTriMesh
        pNewMesh->Attach( pMesh );

    } // End if vertex normals
```

At this point we have a new CTriMesh object whose underlying ID3DXMesh contains the geometry initially loaded from the X file. The next section of code we will look at is also unchanged from the previous lab project.

We start by testing whether this actor (technically, its meshes) is in managed mode or non-managed mode. You will recall from previous lessons that a CTriMesh (or CActor) is automatically placed into non-managed mode by the registration of the CALLBACK_ATTRIBUTEID callback function. If this function is not registered (pointer is NULL), then the mesh should be created in managed mode and we set the local Boolean variable ‘ManageAttribs’ to true. If the callback has been registered, then the callback function pointer will not be NULL and the Boolean variable will be set to false.

After determining a mesh is in managed mode, the following instructs the CTriMesh object to allocate an internal attribute data array to store the textures and materials for each subset. After the CTriMesh::AddAttributeData method is called to allocate the array, we call the CTriMesh::GetAttributeData method to get a pointer to the start of this array. We store this pointer in the local variable pAttribData which will be used later to step through the mesh's attribute array and copy textures and materials. Remember, at this point, the attribute array for the managed mesh is empty.

```
    // Are we managing our own attributes ?
    ManageAttribs =
    (m_pActor->GetCallback( CActor::CALLBACK_ATTRIBUTEID).pFunction == NULL);

    // Allocate the attribute data if this is a manager mesh
    if ( ManageAttribs == true && NumMaterials > 0 )
    {
        if ( pNewMesh->AddAttributeData( NumMaterials ) < -1 )
            { hRet = E_OUTOFMEMORY; goto ErrorOut; }

        pAttribData = pNewMesh->GetAttributeData();

    } // End if managing attributes
```

The next section of code is the 'else' block of the conditional code shown above. It is executed if the mesh is in non-managed mode (i.e., the CALLBACK_ATTRIBUTEID function has been registered with the actor). This code is also unchanged from the previous lab projects. It basically allocates an array that is large enough to hold a global subset ID for each subset in the mesh (pAttribRemap). These global IDs are returned to us by the scene object's attribute callback function. This array is used to re-map the subsets of the mesh to use global IDs instead of local ones. In this section of code however, once the re-map array is allocated, we initialize it with the current subset IDs of the mesh. These may be overwritten with global IDs later, but this is done just so we safely initialize the array.

```
    else
    {
        // Allocate attribute remap array
        pAttribRemap = new ULONG[ NumMaterials ];
        if ( !pAttribRemap ) { hRet = E_OUTOFMEMORY; goto ErrorOut; }

        // Default remap to their initial values.
        for ( i = 0; i < NumMaterials; ++i ) pAttribRemap[ i ] = i;
    } // End if not managing attributes
```

So, if our mesh is in managed mode, we now have a pointer to an empty attribute table that will need to be filled with texture and material information. If the mesh is a non-managed mode mesh, we have a temporary re-map array which we will use to collect global IDs for each of its current subsets.

In the next section of code (also unchanged) we process the materials for each mesh subset. In the managed case, this means extracting the material information (diffuse, specular etc) from the passed material buffer and storing it in the corresponding entry in the managed mesh's internal attribute array. If a CALLBACK_TEXTURE callback function has also been registered by the application, then it will be called to process the texture for each material/subset. Below we see the first section of the material processing loop (handling the managed case):

```
    // Loop through and process the attribute data
    for ( i = 0; i < NumMaterials; ++i )
    {
        if ( ManageAttribs == true )
        {
            // Store material
            pAttribData[i].Material = pMaterials[i].MatD3D;
            pAttribData[i].Material.Ambient = D3DXCOLOR( 1.0f, 1.0f,
                                                         1.0f, 1.0f );

            // Request texture pointer via callback
            Callback = m_pActor->GetCallback( CActor::CALLBACK_TEXTURE);

            if ( Callback.pFunction )
            {
              COLLECTTEXTURE CollectTexture =(COLLECTTEXTURE)Callback.pFunction;

              pAttribData[i].Texture =
              CollectTexture( Callback.pContext, pMaterials[i].pTextureFilename );

                // Add reference. We are now using this
                if ( pAttribData[i].Texture ) pAttribData[i].Texture->AddRef();
            } // End if callback available

        } // End if attributes are managed
```

The above code takes care of populating the managed mesh's attribute array with the texture pointers and material information.

The second part of the loop is the 'else' code block of the conditional which processes each material for non-managed meshes. This code is also unchanged from previous projects.

```
        else
        {
            // Request attribute ID via callback
            Callback = m_pActor->GetCallback( CActor::CALLBACK_ATTRIBUTEID );
            if ( Callback.pFunction )
            {
                COLLECTATTRIBUTEID CollectAttributeID =
                                  (COLLECTATTRIBUTEID)Callback.pFunction;

                AttribID =
                CollectAttributeID( Callback.pContext,
                                    pMaterials[i].pTextureFilename,
                                    &pMaterials[i].MatD3D,
                                    &pEffectInstances[i] );

                // Store this in our attribute remap table
                pAttribRemap[i] = AttribID;

                // Determine if any changes are required so far
                if ( AttribID != i ) RemapAttribs = true;
            } // End if callback available
        } // End if we don't manage attributes
```

```
    } // Next Material
```

We have now processed all the materials. In the case of a managed mesh, the internal attribute table of the mesh now contains all the textures and materials the mesh needs to draw itself. In the non-managed case, we have a temporary array describing the new global ID of each subset. We have not yet applied these global IDs to the mesh data.

In the next section of code we test to see if we were passed any face adjacency information for the mesh by D3DX (which we should have). If so, then we have to copy this face adjacency information into our mesh container's face adjacency array. We never know when we might need the face adjacency information for a mesh, so it is a wise thing to store it in the mesh container just in case. However, we must make a copy of the data and not simply assign the mesh container's member pointer to point straight at it because the adjacency information we have been passed is in memory owned by D3DX and will be destroyed when the function returns. This would leave our mesh container with a dangling pointer. So we allocate a new face adjacency array and copy over the data. Notice how this new array is pointed to by the mesh container's pAdjacency member discussed earlier.

```
    // Copy over adjacency information if any
    if ( pAdjacency )
    {
        pMeshContainer->pAdjacency=new DWORD[ pMesh->GetNumFaces()*3];

        if ( !pMeshContainer->pAdjacency )
        { hRet = E_OUTOFMEMORY; goto ErrorOut; }

        memcpy( pMeshContainer->pAdjacency,
                pAdjacency,
                sizeof(DWORD) * pMesh->GetNumFaces() * 3 );

    } // End if adjacency provided
```

From this point forward, we will start to see the new code that has been added to deal with skinned meshes. Skinned meshes have to be handled very different from regular meshes in a number of different ways, as we will now see.

The next section of code is executed if the mesh we have been passed is a skinned mesh. If we were passed a regular mesh, our job is almost done since the mesh stored in the CTriMesh is pretty much exactly what we want it to be (with the exception of some attribute buffer re-mapping in the non-managed case, which happens towards the end of the function). However, if it is supposed to be a skinned mesh, then the CTriMesh's underlying ID3DXMesh will have to be converted.

The following code tests the ID3DXSkinInfo interface pointer we have been passed by D3DX. If it is not NULL, then it means this ID3DXSkinInfo object contains skinning information for this mesh and that we have encountered a skin in the X file. The first thing we will do in this case is store the ID3DXSkinInfo interface pointer in the mesh container since we will need access to the bone and vertex weight information in many places in our code. We are also careful to increment the reference count when we copy the interface pointer into the mesh container.

```
    // Is this a 'skinned' mesh ?
    if (pSkinInfo != NULL)
    {
        LPD3DXMESH pSkinMesh;

        // first save off the SkinInfo and original mesh data
        pMeshContainer->pSkinInfo = pSkinInfo;
        pSkinInfo->AddRef();
```

We also know from the textbook that the ID3DXSkinInfo object contains the bone offset matrix for every bone used by the skin. We already discussed why we need efficient access to these matrices during rendering which is why we added a bone offset matrix pointer to our mesh container. What we will do is use this pointer to allocate an array of bone offset matrices in the mesh container and copy the bone offset matrices from the ID3DXSkinInfo object into this new array.

```
        // Allocate a set of bone offset matrices, we need to store
        // these to transform any vertices from 'character space' into 'bone space'
        ULONG BoneCount = pSkinInfo->GetNumBones();

        pMeshContainer->pBoneOffset = new D3DXMATRIX[BoneCount];

        if ( !pMeshContainer->pBoneOffset )
            { hRet = E_OUTOFMEMORY; goto ErrorOut; }

        // Retrieve the bone offset matrices here.
        for ( i = 0; i < BoneCount; ++i )
        {
            // Store the bone
            pMeshContainer->pBoneOffset[i] =*pSkinInfo->GetBoneOffsetMatrix(i);

        } // Next Bone
```

The above code first determines how many bones influence this skin, using the ID3DXSkinInfo::GetNumBones method. Since there is a matching bone offset matrix defined for every bone used by the mesh, the number of bones value also tells us how many bone offset matrices there are in the ID3DXSkinInfo object. Once this array is allocated, the code loops through each bone and uses the ID3DXSkinInfo::GetBoneOffsetMatrix method to fetch the bone offset matrix for the corresponding bone offset index. This bone offset matrix is then copied into the mesh container's bone offset matrix array at the same index position. It is very important that we keep a 1:1 mapping between the indices of bone offset matrices stored in our mesh container and their indices in the ID3DXSkinInfo object since this mapping will be essential later when fetching the actual bone matrix pointers from the hierarchy.

In the next section of code we finish off the bulk of the skinned mesh processing through the use of the CAllocateHierarchy::BuildSkinnedMesh helper function. This function (which we will examine next) generates the skinned mesh from the original ID3DXMesh that we currently have. We pass this function the mesh container, the original ID3DXMesh that we have been currently working with, and also the address of an ID3DXMesh interface pointer which on function return will point to a new ID3DXMesh in skinned format. The new mesh (pSkinMesh) will contain vertex weights and possibly vertex matrix indices (in the indexed skinned mesh case). The function will also populate the new members of the mesh container that are necessary during rendering.

Upon return from the BuildSkinnedMesh function, we will have a skinned ID3DXMesh, but it will not be attached to our CTriMesh object. The CTriMesh's underlying ID3DXMesh pointer is still pointing to the original source mesh, so we will replace it so that the CTriMesh now points to the newly generated skin. Notice in the following code that the Attach method of our CTriMesh has an additional Boolean parameter that we set to true. This is because we have added a new behavior to this method. Originally, when we attached a new ID3DXMesh to our CTriMesh object, the attribute table would also be released (for managed meshes). That is, all data that currently existed in the CTriMesh was flushed. However, in this new case, we simply want to replace the geometry (the ID3DXMesh) and leave all the texture and material data intact. Therefore, if true is passed as this parameter, we simply replace the ID3DXMesh pointer and leave the attribute data of the managed mode alone. After all, the skinned mesh will still be using this texture and material data.

```
        // Build the skinned mesh
        hRet = BuildSkinnedMesh( pMeshContainer, pMesh, &pSkinMesh );
        if ( FAILED(hRet) ) goto ErrorOut;

        // pass True to the bReplaceMeshOnly parameter.
        pNewMesh->Attach( pSkinMesh, NULL, true );
        pMesh = pSkinMesh;

    } // End if skin info provided
```

At this point in the code our CTriMesh has all the correct geometry contained and will contain either a skinned or regular ID3DXMesh. The next section of code that we will execute was also found in previous lab projects, but has some additional code to examine.

As in our previous lab projects, one of the last tasks this function has to perform is the re-mapping of the mesh attribute buffer using the global IDs for each subset we compiled earlier. This is only done when the actor/mesh is in non-managed mode. You will recall that we used the attribute callback function to fetch the global IDs for each subset and stored them in the local pAttribRemap array. This array contains an entry for every subset in the original mesh. When we were dealing only with regular meshes, we would simply lock the mesh attribute buffer, loop through each face, and set it to its new global ID. For regular meshes we will still perform this same task. In fact, if we are using software skinning, we will also do the same thing because we are really just using the original source mesh and transforming into a destination mesh ourselves prior to rendering. So in the case of software skinning, the triangles will still be grouped into the same subsets as the original mesh. In the next section of code we test to see if the mesh is a regular or a software skinned mesh and also whether the mesh is in non-managed mode. If so, we adopt the strategy we have all in previous CActor versions and remap the attribute buffer.

```
    // Remap attributes if required
    if ( pAttribRemap != NULL && RemapAttribs == true )
    {
        // Is a skinned mesh ?
        if ( !pSkinInfo ||
             pMeshContainer->SkinMethod == CActor::SKINMETHOD_SOFTWARE )
        {
            ULONG * pAttributes = NULL;

            // Lock the attribute buffer
```

```
            hRet = pMesh->LockAttributeBuffer( 0, &pAttributes );
            if ( FAILED(hRet) ) goto ErrorOut;
            // Loop through all faces
            for ( i = 0; i < pMesh->GetNumFaces(); ++i )
            {
                // Retrieve the current attribute ID for this face
                AttribID = pAttributes[i];

                // Replace it with the remap value
                pAttributes[i] = pAttribRemap[AttribID];

            } // Next Face
```

If the mesh is a software skinned mesh then we must remember that we also have an additional destination mesh stored in the mesh container. This is the mesh that we will be transforming into when we transform the mesh into world space. Therefore, this mesh will also have to have its attribute buffer updated in the same way. After all, we want the subsets in both the source and destination meshes to be the same (the only difference between the two is that one is in model space and the other is in world space). They both use the same textures and materials for each subset, so let us reflect the global subset changes in the destination mesh as well.

The next section of code locks the attribute buffer of the destination mesh and performs a memory copy from the CTriMesh's attribute buffer (which is still locked) into the destination mesh attribute buffer. Remember, at this point we have already updated the source mesh's attribute buffer so we can just copy the attribute buffer directly into the destination mesh such that they are now identical.

```
            // In the software skinning case, we need to reflect any
            // changes made into our SW Mesh
            if ( pSkinInfo &&
                 pMeshContainer->SkinMethod == CActor::SKINMETHOD_SOFTWARE )
            {
                ULONG * pSWAttributes = NULL;

                // Lock the attribute buffer and copy the contents over
                hRet = pMeshContainer->pSWMesh->LockAttributeBuffer(0,
                                                                    &pSWAttributes
                                                                    );
                if ( SUCCEEDED(hRet) )
                {
                    memcpy( pSWAttributes,
                            pAttributes,
                            pMesh->GetNumFaces() * sizeof(ULONG) );
                    pMeshContainer->pSWMesh->UnlockAttributeBuffer( );

                } // End if succeeded
            } // End if software skinning

            // Finish up
            pMesh->UnlockAttributeBuffer( );

        } // End if not skinned
```

Finally, we unlock both the attribute buffers and our mesh is ready to go. However, all we have seen above is the section of the conditional that handles the attribute re-map for non-managed meshes that are either standard meshes or software skins.

In the next section of code we will see the 'else' code block of the conditional which is executed when we are in non-managed mode (as above), but we are using one of the pipeline skinning techniques (Indexed or Non-Indexed Skinning). Why do we need special case code for pipeline skins? Why can't we just alter the attribute buffers for these meshes to contain the new global subset IDs as we did in the regular and software skinning cases?

When we call CAllocateHierarchy::BuildSkinnedMesh, if we have chosen one of the pipeline skinning techniques, the appropriate ID3DXSkinInfo function (ConvertToBlendedMesh or ConvertToIndexedBlendedMesh) will be called to create a new ID3DXMesh which has vertex weights in the vertices. Of course, these functions do much more than just add weights to vertices; they also calculate which bone matrices each subset must use. This information is returned in the bone combination array. These functions may decide that a given subset cannot be rendered because the faces in that subset collectively use more matrix slots than the device has to offer and in this case, the function will often break the subset into multiple subsets, where each uses a smaller set of matrices that the device can handle in hardware. This is where we find our problem…

The returned mesh may have more subsets than the original mesh. In fact, this is usually the case. This is not really a major problem of course, because we are provided with a D3DXBONECOMBINATION structure for each subset in the new mesh. Each structure will contain the original subset ID that this new subset was generated from. This means when we render a managed mode skin, we still have access to the index of the original texture and material in the mesh's internal attribute array that was used by the subset that this new subset was spawned from. However, it is the non-managed mode mesh we are processing now, and we want the scene to be able to batch render this mesh using global subset IDs across mesh boundaries. But what we cannot do is physically change the attribute buffer of the skinned mesh because the local subset IDs are mapped directly to the bone combination table that was returned from the skinned mesh conversion functions.

Of course, all we are really interested in doing is providing a way for the application to call CActor::DrawSubset with the global ID that it generated when the re-map buffer was being compiled and have it all work properly. Since the scene is responsible for setting the texture and material that corresponds to this global ID, all it is really asking is that the actor draw any faces that are connected to this global ID. It is not as if this global ID is used internally by the mesh to set textures and materials as in the managed mesh case, so this is actually very easy to achieve. In fact, it takes less work than in the regular mesh case.

We will not bother altering the values of the mesh attribute buffer at all; internally, all subset IDs will remain zero-based local IDs which map 1:1 with the bone combination table. In the managed mode case, we can use the AttribId member of each bone combination table to index into the texture and materials in the attribute table of the mesh. In the non-managed mode case, we can just store the new global ID here instead:

```
        else
        {
             LPD3DXBONECOMBINATION pBoneComb =(LPD3DXBONECOMBINATION)
                        pMeshContainer->pBoneCombination->GetBufferPointer();

            for ( i = 0; i < pMeshContainer->AttribGroupCount; ++i )
            {
                // Retrieve the current attribute ID for this bone combination
                AttribID = pBoneComb[i].AttribId;

                // Replace it with the remap value
                pBoneComb[i].AttribId = pAttribRemap[AttribID];

            } // Next Attribute Group

        } // End if Skinned mesh

    } // End if remap attributes
```

Basically, we fetch the bone combination table and loop through each of its elements. There will be one element in this array for every subset in the skin. Also remember that this bone combination table will have been created in the CAllocateHierarchy::BuildSkinnedMesh function (which we will cover shortly). Each bone combination structure contains the original subset ID that this new subset was created from. However, in a non-managed mode mesh we no longer need this information since we have no intention of letting the mesh set its own textures and materials. So we will replace the current subset ID with the new global ID compiled for the original subset during the re-mapping process. Our bone combination structures now contain the global ID that was generated for the original (material-based) subset ID which the new (bone-based) subset was generated from. This connection between global IDs and local subsets in a non-managed pipeline skinned mesh and is simple, but may be a little hard to understand at first. Things will fall into place a little more when we actually see this being used in the rendering code later on.

At this point in the code, our CTriMesh is complete. It either holds a regular mesh, a software skin, or one of the pipeline skin types. We no longer need the temporary array we used to store subset re-map information for the non-managed case so we delete it. Also, if we have created a skinned mesh, then the pMesh local interface pointer (that points to our CTriMesh's underlying ID3DXMesh) can also be released as we no longer need it. We release it in the skinned case, because we incremented the skin's reference count earlier in the function when we assigned this local pointer to point at the new skin. Finally, if the CTriMesh is anything other than a software skinned mesh, we perform the usual optimizations (vertex welding and optimize in place). Do you know why we do not perform these optimizations when we have a software skin?

```
    // Release remap data
    if ( pAttribRemap ) delete []pAttribRemap;

    // If this is a skinned mesh, we must release our overriden copy
    // only the CTriMesh will now reference a copy.
    if ( pSkinInfo ) pMesh->Release();

    // Attempt to optimize the new mesh ( unless software skin ).
    if ( !(pSkinInfo && pMeshContainer->SkinMethod==CActor::SKINMETHOD_SOFTWARE) )
```

```
    {
        // Optimize
        pNewMesh->WeldVertices( 0 );
        pNewMesh->OptimizeInPlace( D3DXMESHOPT_VERTEXCACHE );

        // Release the adjacency information, it is no longer valid
        if ( pMeshContainer->pAdjacency ) delete []pMeshContainer->pAdjacency;
        pMeshContainer->pAdjacency = NULL;

    } // End if not software skinned mesh
```

We do not optimize a software skin because, as we know, this will sometimes remove or move vertices and indices inside the vertex and index buffers for better cache coherency and to clean the mesh of any un-used vertices. The problem we have if we do this with a software skinned mesh, is that we need to use the ID3DXSkinInfo object to transform the mesh. This object stores all the vertex index and bone index information which is used by the ID3DXSkinInfo::UpdateSkinnedMesh method. If we start removing vertices from this mesh, all the per-vertex information stored in the ID3DXSkinInfo object will be invalidated and will no longer correctly index the actual vertices inside the mesh. We could implement some re-mapping logic here if we wanted to, but we have decided to live with the limitation in this demo. It is very rare that you will be using the actor for software skinning, so we will just put up with the fact that a software skin will not have its geometry optimized.

The next section of code is virtually unchanged from previous lab projects, although there is one small modification. Like before, it stores our new CTriMesh in the mesh container, but this time what it stores in the mesh container's MESHDATA member varies depending on whether this is a skin. If it is not, then we store a copy of the CTriMesh's underlying ID3DXMesh just as before. However, if it is a skin, then we need to store the original mesh data instead since our CTriMesh subsets have been modified during the skin creation process. This is why we made a copy of the original mesh data pointer at the start of the function. This allows D3DX functions (e.g., D3DXSaveMeshHierarchyToFile) to find the correct mesh data exactly where they expect to find it. In the managed case, we also copy over the texture names and material information into the mesh container's pMaterials array. Once again, this is where D3DX will expect the per-subset material information to be if the actor is saved out to disk. Remember, saving the actor only works with managed mode actors since this is the only mode in which the actor is even aware of the textures and materials that are being used to render it.

```
    // Store our mesh in the container
    pMeshContainer->pMesh = pNewMesh;

    // Store the details so that the save functions have access to them.
    if ( pSkinInfo )
    {
        // Here we store the original mesh, because the actual mesh stored
        // in our CTriMesh has been split up into multiple subsets and prepared
        // for skinning. We need the original data so that we can save the correct
        // data back out.
        pMeshContainer->MeshData.pMesh = pOrigMesh;
        pOrigMesh->AddRef();

    } // End if skinning
    else
    {
```

```
        // When not skinning, we are save to save out our optimized mesh data.
        pMeshContainer->MeshData.pMesh = pNewMesh->GetMesh();

    } // End if not skinning

    // Save the remaining details.
    pMeshContainer->MeshData.Type  = D3DXMESHTYPE_MESH;
    pMeshContainer->NumMaterials   = NumMaterials;

    // Copy over material data only if in managed mode so we can save X file)
    if ( NumMaterials > 0 && ManageAttribs == true )
    {
        // Allocate material array
        pMeshContainer->pMaterials = new D3DXMATERIAL[ NumMaterials ];

        // Loop through and copy
        for ( i = 0; i < NumMaterials; ++i )
        {
            pMeshContainer->pMaterials[i].MatD3D = pMaterials[i].MatD3D;
            pMeshContainer->pMaterials[i].pTextureFilename
                           = _tcsdup( pMaterials[i].pTextureFilename );

        } // Next Material

    } // End if any materials to copy
```

Finally, we assign the passed mesh container pointer to point at our new mesh container structure so that it will be accessible to D3DX on function return.

```
    // Store this new mesh container pointer
    *ppNewMeshContainer = (D3DXMESHCONTAINER*)pMeshContainer;

    // Success!!
    return D3D_OK;

ErrorOut:

    // If we drop here, something failed
    DestroyMeshContainer( pMeshContainer );

    if ( pDevice      ) pDevice->Release();
    if ( pAttribRemap ) delete []pAttribRemap;
    if ( pNewMesh     ) delete pNewMesh;

    // Failed....
    return hRet;
}
```

At the bottom of the function you see the section of code that is jumped to when an error occurs. It simply cleans up any memory being used when the error occurred before returning failure.

CAllocateHierarchy::BuildSkinnedMesh

A new function that has been added to our CAllocateHierarchy class is the BuildSkinnedMesh method. We learned while covering the previous function that this function is called to convert the regular ID3DXMesh into a new skinned mesh. The function accepts three parameters. The first parameter is a pointer to the mesh container in which the skin information will be stored. Remember, there are many other new members in our mesh container now that hold information about the skin; this function will assign these members the correct values for the type of skin created. The second parameter is a pointer to an ID3DXMesh containing the regular mesh which needs to be converted into a skin. We saw in the last function, that for this parameter we pass in the ID3DXMesh that was loaded from the X file and assigned to the CTriMesh. As the final parameter, we pass the address of an ID3DXMesh interface pointer which will be assigned the address of the new skinned mesh we create. When this function returns the skinned mesh to the calling function, the original mesh is removed from the CTriMesh and the new skinned mesh (pMeshOut) will be attached in its place. The original mesh can then be deleted.

Let us now have a look at this method a section at a time. The function first retrieves the device being used by the actor so that we can retrieve the capabilities of the device to test it for the varying levels of skinning support. Once we have retrieved the capabilities, we release the device interface since we no longer need it.

```
HRESULT CAllocateHierarchy::BuildSkinnedMesh
                                    ( D3DXMESHCONTAINER_DERIVED * pMeshContainer,
                                      LPD3DXMESH pMesh,
                                      LPD3DXMESH * pMeshOut )
{
    D3DCAPS9            Caps;
    HRESULT             hRet;
    ULONG               i, j, k;
    LPD3DXSKININFO      pSkinInfo  = pMeshContainer->pSkinInfo;
    LPDIRECT3DDEVICE9   pDevice    = NULL;

    // Retrieve the Direct3D Device and poll the capabilities
    pDevice = m_pActor->GetDevice();
    pDevice->GetDeviceCaps( &Caps );
    pDevice->Release();
```

In the next line of code we call the other new member function of our CAllocateHierarchy class, called DetectSkinningMethod. This function is a very simple utility that returns one of the following:

SKINMETHOD_INDEXED
SKINMETHOD_NONINDEXED
SKINMETHOD_SOFTWARE

If the current skinning method for the actor has been set to anything other than SKINMETHOD_AUTODETECT, this detection function will simply return the mode that was specified. In other words, the function returns the skin method chosen by the application for the actor. However, if the actor's skinning method has been set to SKINMETHOD_AUTODETECT, this function will query the hardware to find the best skinning method available. This will be either SKINMETHOD_INDEXED or SKINMETHOD_NONINDEXED.

```
    // Calculate the best skinning method and store
    pMeshContainer->SkinMethod = DetectSkinningMethod( pSkinInfo, pMesh );
```

At this point, we now have the skinning method that will be used for this mesh (stored in our mesh container's SkinMethod member). The remainder of the BuildSkinnedMesh function is divided into three sections that create the mesh as either an indexed skin, a non-indexed skin, or a software skin. The first section creates an indexed skinned mesh if that is what has been requested.

The first thing we do in the indexed case is fetch the index buffer of the source mesh so that we can pass it into the ID3DXSkinInfo::GetMaxFaceInfluences function. This function will use the index buffer to determine the maximum number of bones that influence a given triangle in the mesh. We need to know this because the device must have at least this many matrix slots, otherwise the indexed skin will not be able to be transformed with hardware vertex processing. Remember, the ConvertToIndexedBlendedMesh function will subdivide the mesh into smaller subsets if necessary so that each subset can be transformed with the number of matrix slots available, even if it has to break the mesh down into lots of one triangle subsets. However, if there are one or more triangles in this mesh that have more influences than the device has matrix slots, then this mesh can not be transformed by the hardware (not even if it is broken down into one triangle subsets). In such a case, we will have to revert to software vertex processing when rendering this mesh.

```
    // Built differently based on the type of skinning required
    if ( pMeshContainer->SkinMethod == CActor::SKINMETHOD_INDEXED )
    {
        ULONG MaxFaceInfluences;
        ULONG Flags = D3DXMESHOPT_VERTEXCACHE;

        LPDIRECT3DINDEXBUFFER9 pIB;
        hRet = pMesh->GetIndexBuffer(&pIB);
        if (FAILED(hRet)) return hRet;

        // Retrieve the maximum number of influences for the mesh index buffer
        hRet = pSkinInfo->GetMaxFaceInfluences( pIB,
                                                pMesh->GetNumFaces(),
                                                &MaxFaceInfluences);
        pIB->Release();
        if (FAILED(hRet)) return hRet;
```

In the above code, we fetch the maximum number of influences for a given face in the mesh and store that in the MaxFaceInfluences local variable. We then release the index buffer interface we fetched earlier since we no longer need it.

We now know the maximum number of influences a single face uses in the mesh and we know that if this is larger than the number of matrices supported by the device in hardware then this mesh must be rendered using indexed skinning. Actually, that is not quite true! It is possible in some cases that the number of influences for a given face could actually be larger than 12. However, as the pipeline only supports four influences per vertex (i.e., 12 per face), we know that if this is the case, the ConvertToIndexedBlendedMesh function will simply ignore any per-vertex influences over the maximum of four. Even if we have skinning information that tells us a face is influenced by more than 12 matrices, the ConvertToIndexedBlendedMesh function would downgrade the resulting mesh so that

at most there would only ever be 12 influences per face. We are going to compare the MaxFaceInfluences member we just calculated against the number of matrices available on the device to see if the resulting indexed mesh can be rendered in hardware. The problem is, if MaxFaceInfluences is higher than 12, we will fail the hardware case and drop into software mode, even though the resulting mesh will have at most 12 influences per face.

For example, imagine that we have a device that supports 12 matrices and that we have determined that the maximum number of influences per face is 20. We would compare 12 against 20 and find that the maximum face influences is larger than the number of matrices available. In this case we would fail and fall back to software vertex processing. However, this would technically be unnecessary because we know that ConvertToIndexedBlendedMesh will drop those surplus influences as no face can ever have more than 12 influences. Thus, the generated skinned mesh would have a maximum face influence count of 12. We have 12 matrices on our example device matrix palette so this mesh could have been rendered in hardware. Therefore, we must take this into account when performing the comparison against the number of matrices available on the device.

In order to do this we modify the MaxFaceInfluences value so that it can never be larger than 12. We do this by assigning MaxFaceInfluences the minimum between its current value and 12. If maximum face influences is less than 12, its value will be unchanged; if the maximum face influence is greater than 12, it will be assigned 12. As long as the device supports 12 matrices, we will always be able to render the indexed skin in hardware, even if it is broken into one triangle subsets.

```
        MaxFaceInfluences = min( MaxFaceInfluences, 12 );
```

Next we fetch the number of matrices the device supports in hardware. The device capability member that tells us this is the MaxVertexBlendMatrixIndex cap. This is the highest matrix index that the device supports in hardware. Since this is a zero-based number, we know that if this value is 31 for example, the device supports 32 matrices (matrices 0 – 31). To calculate the number of matrices supported by the device, we simply add one to this value.

```
        ULONG MaxSupportedIndices = Caps.MaxVertexBlendMatrixIndex + 1;
```

Next we test to see if the mesh has normals. If it does then it means lighting is being used and the normals must also be transformed. When this is the case, half of the matrix slots will be taken up with matrices to transform the normals. Thus, we actually lose half the number of bones we can transform simultaneously. For example, if the device supports 256 matrices but the vertices have normals, the driver will need to keep reserve half that amount (128) for storing additional copies of the matrix (the inverse transpose of each bone matrix) to transform the normals. While we do not have to set these normal matrices ourselves, we have to be aware that the hardware will need space for them and take this into account. In the next line of code we divide the number of available hardware matrix slots the device can use if normals are present.

```
        if ( pMesh->GetFVF() & D3DFVF_NORMAL) MaxSupportedIndices /= 2;
```

Finally, we test to see if the number of matrices supported by the device (MaxSupportedIndices) is smaller than the maximum number of influences a single face uses. If it is, then the mesh has one or

more triangles that cannot be transformed using hardware vertex processing. In the following code snippet, you can see that when this is the case, we set the mesh container's SoftwareVP member to true so that we know during rendering we must enable software vertex processing before rendering this mesh. We also combine the local Flags member with the D3DXMESH_SYSTEMMEM flag. These flags will be used in a moment to describe the resource pool we would like the skinned mesh to be created in. Since this is going to be a software vertex processing mesh, we want the mesh to be created in system memory so that the CPU can efficiently access the data.

```
        if ( MaxSupportedIndices < MaxFaceInfluences )
        {
            // HW does not support indexed vertex blending. Use SW instead
            pMeshContainer->PaletteEntryCount = min(256,
                                                    pSkinInfo->GetNumBones());
            pMeshContainer->SoftwareVP = true;
            Flags |= D3DXMESH_SYSTEMMEM;
        }
```

In the above code you can also see that we store the number of matrices that will be used for rendering in the mesh container's PaletteEntryCount. Because this is the software vertex processing case, we always have 256 matrix slots on the device to call upon. However, this value will be passed into the ConvertToIndexedBlendedMesh function and will ultimately decide the size of each subset's BoneId array in the returned bone combination table. Certainly we do not want every subset to have a 256 entry BoneId array if we never use more than 20 matrices for a given subset. This is very important, since we will have to loop through and test for valid bone IDs during the rendering of a subset. There is no point in testing every element of a 256 element bone ID array when we only use a handful of bones per subset. So, although we can technically use up to 256 bone matrices, we will never need more matrix slots than the number of bones in the skin. Therefore, the palette size is calculated to be only as large as we need it to be (i.e., either 256 or the number of bones in the skinned mesh, whichever is smaller). When this value is passed to the CovertToIndexedBlendedMesh function, the BoneID array of each subset will contain this many elements. (Of course, each subset may not use all those elements.)

In the following code, we see the 'else' block to the above conditional. It gets executed if the hardware matrix palette is large enough to cater for the the maximum number of matrices that influence a given face.

```
        else
        {
            pMeshContainer->PaletteEntryCount = min( MaxSupportedIndices,
                                                     pSkinInfo->GetNumBones() );
            pMeshContainer->SoftwareVP = false;
            Flags |= D3DXMESH_MANAGED;
        }
```

Do not forget that MaxSupportedIndices is the number of matrix slots available on the device, which may have already been divided in half if normals are being used. We then set the mesh container's SoftwareVP member to false so that we know we can render this mesh in hardware. As this is a hardware vertex processing mesh, we add the D3DXMESH_MANAGED flag to the flags member so that the mesh we create will be allocated in the managed resource pool (video memory).

We have now done all the preparation work for creating the indexed skinned mesh. All that is left to do is create the mesh itself using the ID3DXSkinInfo::ConvertToIndexedBlendedMesh function:

```
        // Convert the mesh
        pSkinInfo->ConvertToIndexedBlendedMesh( pMesh,
                                                Flags,
                                                pMeshContainer->PaletteEntryCount,
                                                pMeshContainer->pAdjacency,
                                                NULL, NULL, NULL,
                                               &pMeshContainer->InfluenceCount,
                                               &pMeshContainer->AttribGroupCount,
                                               &pMeshContainer->pBoneCombination,
                                                pMeshOut );

    } // End if Indexed Skinning
```

The first parameter is the source ID3DXMesh that was passed into the function. This will contain the original geometry that was loaded from the X file and is the mesh we wish to clone into a skinned mesh.

The second parameter contains the mesh creation flags which we have seen many times before. As we just discussed, if we have determined we need to software vertex process this mesh, these flags will specify that the new mesh should be created in the system memory resource pool. If the hardware can transform and render this mesh, the flags will instruct the function to create the new mesh in the managed resource pool.

The third parameter is the palette size we calculated earlier. This tells the function how many matrix slots it can utilize and influences how the function breaks the mesh up into subsets. The larger the palette size, the better chance we have of subsets not being split into smaller subsets based on bone combination. However, as discussed above, this should not be larger than is needed.

The fourth parameter is the adjacency information for the source mesh which was calculated and stored in the mesh container prior to this function being called (we were passed this array by D3DX in the CreateMeshContainer method).

We pass NULL as the next three parameters since we are not interested in receiving the face adjacency information or the face and vertex re-map information of the output mesh (the skin).

As the eighth parameter, we pass the address of our mesh container's InfluenceCount member, which the function will fill with the maximum number of bones that influence a given **vertex**. This tells us how many weights will be in the vertex format of the output mesh and is used during rendering to set the vertex blending render state to allow the pipeline to account for the correct number of weights in each vertex it transforms.

The ninth parameter is the address of the mesh container's AttributeGroupCount member. On function return, it will contain the number of subsets in the new output mesh. This will also describe the number of D3DXBONECOMBINATION structures in the bone combination table.

The tenth parameter is the address of our mesh container's ID3DXBuffer interface pointer (pBoneCombination). On function return, it will contain an array of D3DXBONECOMBINATION structures; one for each subset in the new mesh. Each element in this array will describe the bone and bone offset matrices used by the subset. We must set these bone matrices on the device prior to rendering the subset.

As the final parameter, we pass in the address of an ID3DXMesh interface pointer which, on successful function return, will point to our new skinned mesh.

At this point our indexed skinned mesh has been successfully created and there is nothing else to do in the indexed skinned case.

We will now discuss the section of code that is executed if a non-indexed skinned mesh is to be created. This will be the case if the appliction specifically set the actor to use SKINMETHOD_NONINDEXED or if the auto detection function returned SKINMETHOD_NONINDEXED as the most efficient skinning method supported by the hardware.

In the non-indexed case, creating the mesh is a lot easier. In the following code you can see that we immediately call ID3DXSkinInfo::ConvertToBlendedMesh to create the mesh and we do not have to worry about calculating palette sizes or anything like that. The function is passed exactly the same parameters as in the non-indexed case.

```
    else if ( pMeshContainer->SkinMethod == CActor::SKINMETHOD_NONINDEXED )
    {
       ULONG CurrentIndex = 0;
       // Convert the mesh
       pSkinInfo->ConvertToBlendedMesh(pMesh,
                                       D3DXMESH_MANAGED | D3DXMESHOPT_VERTEXCACHE,
                                       pMeshContainer->pAdjacency,
                                       NULL, NULL, NULL,
                                       &pMeshContainer->InfluenceCount,
                                       &pMeshContainer->AttribGroupCount,
                                       &pMeshContainer->pBoneCombination,
                                        pMeshOut );
```

Once thing to watch out for in the above code is the value returned in the mesh container's InfluenceCount member since it has a slightly different meaning than in the indexed case. In the indexed case, this is the maximum number of bones that influence a given vertex in the mesh. It can be used during rendering to set the D3DRS_VERTEXBLEND render state correctly. In the non-indexed case, this value will be set to contain the maximum number of bones that influence a given **subset**. It is implicitly also the palette size (which is 4 at most). In the non-indexed case, InfluenceCount describes the size of each subset's BoneId array, while in the indexed case, the palette size member contained this value.

At this point we have created our new non-indexed mesh. However, unlike the indexed case, where the mesh is either in hardware or software vertex processing mode, in the non-indexed case it is possible to have a mixture of subsets -- some of which can be rendered in hardware and some which cannot. So our next task will be to find which subsets use more bone influences than the device can handle (4 at most)

and store them as subsets that need to be rendered with software vertex processing enabled. You will recall from our earlier discussions that this will involve stepping though the bone combination structure in two passes. In the first pass we will find all hardware capable subsets and store them in our mesh container's pVPRemap array. We will then take another pass though the subsets searching for any software-only subsets and add them to the pVPRemap array as well.

At the end of this process, the pVPRemap array will contain the indices of all hardware capable subset IDs followed by all the software-only subset IDs. We will have them stored in two batches that can be efficiently rendered without having to perform these tests each time we render. Before we render the batch of software-only subsets, we will have to enable software vertex processing.

So how do we determine which subsets in our new skinned mesh are hardware supported? Well, first we fetch a pointer to the bone combination array that was returned from the ConvertToBlendedMesh function call. We then allocate the subset re-map array to store a number of elements equal to the number of subsets in the new mesh:

```
        LPD3DXBONECOMBINATION pBoneComb = (LPD3DXBONECOMBINATION)
                              pMeshContainer->pBoneCombination->GetBufferPointer();

         // Allocate space for the vertex processing remap table
         pMeshContainer->pVPRemap = new ULONG[ pMeshContainer->AttribGroupCount ];
         if ( !pMeshContainer->pVPRemap )
             { (*pMeshOut)->Release(); return E_OUTOFMEMORY; }
```

Now we have a pointer to the bone combination table (one element for each subset) and an empty array of ULONG's which we are about to populate with the mesh's subset IDs in hardware/software order.

We will need to loop through each subset twice. In the first pass through the outer loop we will be searching for hardware subsets and adding their IDs to our re-map array. In the second iteration of the loop we will be adding software subsets.

```
        // Loop through each of the attributes and determine how many bones
        // influence the vertices in this set, and determine whether it is
        // software or hardware.
         for ( i = 0; i < 2; ++i )
         {
             // First pass is for hardware, second for software
             for ( j = 0; j < pMeshContainer->AttribGroupCount; ++j )
             {
```

Our next task will be to fetch the bone combination structure for the current subset being processed and check each element in its BoneId array to see if it holds a valid index (UNT_MAX == invalid). As discussed earlier, the mesh container's InfluenceCount member now holds the number of elements in each BoneId array, so we will use this to loop through each element. For every valid bone ID we find, we will increment the local loop variable InfluenceCount to keep a running tally for the current subset of how many bones it is influenced by.

```
                ULONG InfluenceCount = 0;
```

```
                // Loop through for each of our maximum 'influence' items
                for ( k = 0; k < pMeshContainer->InfluenceCount; ++k )
                {
                   // If there is a bone used here, increase our max influence
                    if ( pBoneComb[ j ].BoneId[ k ] != UINT_MAX) InfluenceCount++;

                } // Next influence entry
```

At the end of the inner loop shown above, we have calculated the number of bones needed to render the current subset being processed and stored the result in the local InfluenceCount variable. We now test this value against the device capability MaxVertexBlendMatrices, which tells us how many of the possible 1-4 matrices used by the non-indexed skinning technique are supported by the device. If this subset has more influences than the hardware can support then we have found a subset that must be rendered with software vertex processing. Therefore, if we are on the second pass (i == 1) of the outer loop (the pass that searches for software subsets) we add this subset's index to the re-map array. Alternatively, if we have found a subset that can be rendered in hardware and we are on the hardware search pass (i == 0) we add the subset ID to the re-map array.

However, notice that in the first pass (the hardware pass), whenever we add a subset to the re-map array, we also increment the mesh container's SWRemapBegin variable. The idea is that at the end of the first pass, this value will contain the index into the re-map array where the block of software subsets begin. This allows us to easily render the hardware and software subsets in two blocks during the rendering traversal of the hierarchy.

```
                if ( InfluenceCount > Caps.MaxVertexBlendMatrices )
                {
                    // If it exceeds, we only store on the software pass
                    if ( i == 1 ) pMeshContainer->pVPRemap[ CurrentIndex++ ] = j;

                } // End if exceeds HW capabilities
                else
                {
                    // If it is within HW caps, we only store on the first pass
                    if ( i == 0 )
                    {
                        pMeshContainer->pVPRemap[ CurrentIndex++ ] = j;
                        pMeshContainer->SWRemapBegin++;

                    } // End if HW pass

                } // End if within HW capabilities

            } // Next attribute group

            // If all were found to be supported by hardware, no need to test for SW
            if ( pMeshContainer->SWRemapBegin >= pMeshContainer->AttribGroupCount )
                break;

        } // Next Pass
```

Note at the bottom of the outer loop (the pass loop) that we test to see if the SWRemapBegin member of the mesh container is equal to the number of subsets in the mesh. If this test is performed at the end of

the first pass (the hardware pass) and is found to be true, it means that all subsets can be hardware rendered so there is no need to execute the second pass because no software subsets exist. When this is the case, we simply break from the loop instead of continuing on to the second iteration of the loop.

At this point, we have created our non-indexed mesh and determined which subsets need to be hardware or software rendered. Our job is almost done except for one small matter. If we found that some of the subsets needed to be rendered in software, then it means that we need to enable software vertex processing before we render those subsets. But in order for us to render a mesh in the managed resource pool using software vertex processing, the mesh must have been created with the D3DXMESH_SOFTWAREPROCESSING flag. You can be sure that the ConvertToBlendedMesh function did not create our mesh with this flag, so if software subsets are needed, we must clone our skinned mesh so that we can add this new flag. This is what the next section of code does. It clones our skinned mesh into a new skin capable of software vertex processing. We then release the original skin and use the clone instead. After that, our job is done; we have successfully created our non-indexed skinned mesh.

```
        // If there are entries which require software processing, we must clone
        //  our mesh to ensure that software processing is enabled.
        if ( pMeshContainer->SWRemapBegin < pMeshContainer->AttribGroupCount )
        {
            LPD3DXMESH pCloneMesh;

            // Clone the mesh including our flag
            pDevice = m_pActor->GetDevice();
           (*pMeshOut)->CloneMeshFVF( D3DXMESH_SOFTWAREPROCESSING |
                                      (*pMeshOut)->GetOptions(),
                                      (*pMeshOut)->GetFVF(),
                                      pDevice,
                                      &pCloneMesh );
            pDevice->Release();

            // Validate result
            if ( FAILED( hRet ) ) return hRet;

            // Release the old output mesh and store again
            (*pMeshOut)->Release();
            *pMeshOut = pCloneMesh;

        } // End if software elements required

    } // End if non indexed skinning
```

The next and final section of code is executed if a pure software skinned mesh is to be used. The only way this case can ever be executed is if the application specifically set the actor's skinning method to SKINMETHOD_SOFTWARE since the auto detection code will never fall back to this mode. This is because the auto-detection method favors the pipeline skinning techniques using software vertex processing over pure software skinning.

If we are using software skinning, then we already have the mesh created. The source mesh passed into the function (which is currently attached to our CTriMesh) already contains the model space vertices of the skin in its reference pose, and there is nothing else we have to do to it. However, we will need an

additional mesh (a destination mesh) that we can use to transform our geometry into prior to rendering. Therefore, we simply clone the current mesh to create a copy and assign it to the mesh container's pSWMesh member. While this mesh currently contains a copy of the vertex data in the source mesh, the vertex data itself will be overwritten as soon as we call ID3DXSkinInfo::UpdateSkinnedMesh in response to a hierarchy update. However, by cloning it in this way, our destination mesh will have the same index and attribute buffer as the source mesh, which is exactly what we want. After all, it is only the vertex buffer of the destination mesh that will be overwritten each time the mesh is updated; the index and attribute information should be identical for both meshes.

```
    else if ( pMeshContainer->SkinMethod == CActor::SKINMETHOD_SOFTWARE )
    {
        // Clone the mesh that we'll be using for rendering software skinned
        pDevice = m_pActor->GetDevice();

        hRet = pMesh->CloneMeshFVF( D3DXMESH_MANAGED,
                                    pMesh->GetFVF(),
                                    pDevice,
                                    &pMeshContainer->pSWMesh );
        pDevice->Release();

        // Validate result
        if ( FAILED( hRet ) ) return hRet;

        // Add ref the mesh, we're going to pass it straight back out
        pMesh->AddRef();
        *pMeshOut = pMesh;

    } // End if Software Skinning

    // Success!!
    return D3D_OK;
}
```

Finally, we now have covered the two largest and arguably most intimidating functions in this lab project. These are the functions that load the mesh, populate the mesh container, and generate a skinned mesh in any of our supported flavors. But we do still have one gap in the coverage thus far -- at the very top of the BuildSkinnedMesh function, a call was made to CAllocateHierarchy::DetectSkinningMethod. This function determines which skinning method the actor should used based on the current skinning method of the actor set by the application and the capabilities of the hardware (if the actor is in the default auto detect mode).

CAllocateHierarchy::DetectSkinningMethod

This method is called at the top of the BuildSkinnedMesh function to determine the skinned mesh creation strategy that should be employed. The first part of the function simply tests to see whether auto-detection of a skinning method is even required. The default skinning method of the actor is SKINMETHOD_AUTODETECT (set in the constructor), which means the function should try to determine the most efficient skinning method for the current hardware. However, the application can change the skinning mode of the actor prior to the load call to specifically state that that it would like to

use indexed, non-indexed, or software skinning. In such cases, this function has no work to do since auto-detection is not required. The function will simply return the current skin mode of the actor as set by the application.

In the first section of this function we retrieve the skinning mode of the actor. If it set to anything other than SKINMETHOD_AUTODETECT, then auto detection is not required and this function can simply return the actor's skinning method back to BuildSkinnedMesh:

```
ULONG CAllocateHierarchy::DetectSkinningMethod( ID3DXSkinInfo * pSkinInfo,
                                                LPD3DXMESH pMesh ) const
{
    HRESULT hRet;
    ULONG   SkinMethod = m_pActor->GetSkinningMethod();

    // Autodetecting?
    if ( !(SkinMethod & CActor::SKINMETHOD_AUTODETECT) )
    {
        // We are not to autodetect, so just return it
        return SkinMethod;

    } // End if no auto-detect
```

If the SKINMETHOD_AUTODETECT method is chosen, then the 'else' block of this conditional will be executed (shown below). The rest of this function simplifies the auto-detection process by first finding out if both indexed skinning and non-indexed skinning are supported in hardware for this mesh. Once we have compiled this information, we can decide which one to use based on any modifier flags that may have been passed to influence the process.

We initially create two local boolean variables called bHWIndexed and bHWNonIndexed which will be initially set to false (meaning none are supported). We will then perform the appropriate tests for each skinning technique and set these variables to true if we find the mesh is supported in hardware by these skinning techniques.

In the first section of the 'else' block we get the capabilities of the current device being used by the actor. Our first test will be to see if hardware indexed skinning is supported for our mesh.

```
    else
    {
        LPDIRECT3DINDEXBUFFER9  pIB;
        D3DCAPS9                Caps;
        LPDIRECT3DDEVICE9       pDevice;
        ULONG                   MaxInfluences;
        bool                    bHWIndexed = false, bHWNonIndexed = false;

        // Retrieve the Direct3D Device and poll the capabilities
        pDevice = m_pActor->GetDevice();
        pDevice->GetDeviceCaps( &Caps );
        pDevice->Release();

        // First of all we will test for HW indexed support
        bHWIndexed = false;
```

We will get the index buffer of the mesh and pass it into the ID3DXSkinInfo::GetMaxFaceInfluences function. This will return (in the MaxInfluences local variable) the maximum number bones influencing a single triangle in the mesh. If this number is greater than the number of hardware matrix slots on the device, hardware indexed skinning is not supported for this mesh. If the maximum face influences is less than or equal to the number of matrix device slots, then we can perform hardware indexed skinning, so we set the bHWIndexed boolean to true.

```
        // Retrieve the mesh index buffer
        hRet = pMesh->GetIndexBuffer(&pIB);
        if ( !FAILED(hRet) && Caps.MaxVertexBlendMatrixIndex > 0  )
        {
            // Retrieve the maximum number of influences for the mesh index buffer
                    pSkinInfo->GetMaxFaceInfluences( pIB,
                                                     pMesh->GetNumFaces(),
                                                     &MaxInfluences );

            pIB->Release();


             if (Caps.MaxVertexBlendMatrixIndex + 1 >= MaxInfluences )
                 bHWIndexed = true;

        } // End if no failure
```

Now we will test to see if this mesh can be rendered in hardware using non-indexed skinning. We use the ID3XSkinInfo::GetVertexInfluences method to retrieve the maximum number of bones that influence a single vertex in the mesh. This will be a maximum of 4 in the non-indexed case. If we have fewer matrices to use than this, then it means there are vertices in this mesh that require more matrices/weights than are supported by the hardware. For example, if the maximum vertex influence is 4, then we need all four matrix slots to be used to transform this mesh in hardware. If we only have two matrix slots on the device, then hardware non-indexed support for this mesh is not available. If this is not the case and we do have enough matrices, we set the Boolean variable bHWNonIndexed to true.

```
        // Now we will test for HW non-indexed support
        bHWNonIndexed = false;

        // It should be safe to assume that if the maximum number of vertex
        // influences is larger than the card capabilities, then at least one
        // of the resulting attribute groups generated by ConvertToBlendedMesh
        // will require SoftwareVP
        hRet = pSkinInfo->GetMaxVertexInfluences( &MaxInfluences );

        // Validate
        if ( !FAILED(hRet) && Caps.MaxVertexBlendMatrices >= MaxInfluences )
              bHWNonIndexed = true;
```

At this point, we now know which skinning techniques are supported in hardware (none, one, or both). All we have to do now is choose the best one. In the next section of code you can see that if hardware indexed support is available and non-indexed hardware support is not available, then our job is easy -- we want to use the indexed skinning method. Even if both were supported in hardware this would be preferable.

```
        // Return the detected mode
        if ( bHWIndexed == true && bHWNonIndexed == false )
        {
            // Hardware indexed is supported in full
            return CActor::SKINMETHOD_INDEXED;

        } // End if only indexed is supported in full in hardware
```

If this is not the case, then we try the opposite test to see if non-indexed is supported in hardware but indexed is not. Once again, this is an easy choice for us since we are always going to favor the hardware option.

```
        else if ( bHWNonIndexed == true && bHWIndexed == false )
        {
            // Hardware non-indexed is supported in full
            return CActor::SKINMETHOD_NONINDEXED;

        } // End if only non-indexed is supported in full in hardware
```

If we get this far, then it might mean that both skinning types are supported in hardware and we have to make a choice. Usually, we will always choose indexed over non-indexed skinning. However we discussed earlier that when we set the actor's skinning method, we can also specify the SKINMETHOD_PREFER_HW_NONINDEXED modifier flag. If this flag is set, it means that the application would like to favor hardware non-indexed skinning over hardware indexed skinning when both are available. So in the next section of code, you can see that if both hardware skinning methods are available, we return SKINMETHOD_INDEXED unless this modifier flag has been set. If it is, we return SKINMETHOD_NONINDEXED instead.

```
        else if ( bHWNonIndexed == true && bHWIndexed == true )
        {
            // What hardware method do we prefer since they are both supported?
            if ( SkinMethod & CActor::SKINMETHOD_PREFER_HW_NONINDEXED )
                return CActor::SKINMETHOD_NONINDEXED;
            else
                return CActor::SKINMETHOD_INDEXED;
        }
```

Finally, if we get this far without returning a skinning method, it means none of the pipeline skinning methods are supported in hardware. Now we will need to choose which one to use with software vertex processing. Again, remember that auto-detection always returns a pipeline skinning technique and never returns pure software skinning mode.

It is difficult to make a blanket statement about which skinning technique will be more efficient when using software vertex processing. Indexed skinning allows much better batch rendering potential, but then again, the multiplication of redundant matrices with zero weights could really hurt performance now that those matrix multiplications are being done on the CPU. Non-indexed skinning can often result in smaller subsets, but it also benefits from being able to set the D3DRS_VERTEXBLEND render state on a per-subset basis during rendering.

For our purposes, we make a design decision that says that the default auto-protection process would choose software non-indexed skinning over software indexed skinning. However, the application can specify the SKINMETHOD_PREFER_SW_INDEXED modifier flag if desired. If this flag is not set, we will return SKINMETHOD_NONINDEXED; otherwise we will return SKINMETHOD_INDEXED instead. Below we see the remainder of the function:

```
        else
        {
            // What software method do we prefer since neither are supported?
            if ( SkinMethod & CActor::SKINMETHOD_PREFER_SW_INDEXED )
                return CActor::SKINMETHOD_INDEXED;
            else
                return CActor::SKINMETHOD_NONINDEXED;

        } // End if both provide only software methods

    } // End if auto detect
}
```

We have now covered the complete hierarchy and mesh creation process for a CActor class which supports skinned meshes. All of the functions we covered in this section are executed when the actor calls the D3DXLoadMeshHierarchyFromX function. So let us now have a look at the CActor::LoadActorFromX function to see if anything has changed. This is the function that is called from the application to load an X file into an actor.

CActor::LoadActorFromX

We know that CActor::LoadActorFromX calls the D3DXLoadMeshHierarchyFromX function to load X files. We also know that the D3DX function will call CAllocateHierarchy::CreateMeshContainer for every mesh found in the file. Therefore, all of the functions we have discussed earlier (CreateMeshContainer, BuildSkinnedMesh, and DetectSkinningMethod) are executed at this time; once for each mesh in the file. When D3DXLoadMeshHierarchyFromX returns program flow back to CActor::LoadActorFromX we might assume then that our work is done and no changes to this function would need to be made. While this is mostly true, some small modifications will be required.

After the hierarchy and its meshes have been loaded, any skinned mesh containers that exist in the hierarchy will not yet have all the information they need. You will recall that our mesh container structure stores an array of bone matrix pointers and an array of corresponding bone offset matrices. We saw in the CreateMeshContainer method how we populated the mesh container's bone offset array without any difficulty -- we extracted the bone offset matrices that were stored in the ID3DXSkinInfo object and copied them into the mesh container. We can do this for bone offset matrices because they are stored inside the file and never change. However, we have not yet copied all the bone matrix pointers from the hierarchy into the mesh container's bone matrix pointer array. Why did we not do this at the same time as we copied over the bone offset matrices? The reason is quite simple…

During hierarchy construction, the frame hierarchy has not been fully loaded yet, so we might not have access to all of the frames used as bones by the skin. For example, imagine a skinned mesh uses five bones and its mesh container is attached to the first (parent) bone. When D3DX is loading the hierarchy, it will do so one frame at a time. It will call CAllocateHierarchy::CreateFrame first to allow our callback to allocate the frame structure. Next it will test to see if a mesh is present, and if so, will call CAllocateHierarchy::CreateMeshContainer. Herein we find our problem; the mesh we are creating uses five frames as bones and yet four of those bones have not even been loaded or processed yet. So how can we possibly copy all the bones into the mesh's bone matrix array when the skeleton has not yet been fully loaded? Of course, we cannot. That is why we must collect the bone matrices for each mesh as a final step *after* the entire hierarchy has loaded and D3DXLoadMeshHierarchyFromX has returned program flow back to us.

The CActor::LoadActorFromX function has had a single function call added to handle this task. It is called CActor::BuildBoneMatrixPointers and it will be invoked right after the loading function returns. We will look at this function in a moment, but its job is very simple. For each mesh container in the hierarchy, it finds the matching bone matrix in the hierarchy for each bone offset matrix (these are linked by bone name) stored in the mesh container. It then copies that bone matrix pointer into the array in the mesh container. Remember, the bone matrices are the absolute matrices of the frames in the hierarchy being used as bones by the skin. At the end of the BuildBoneMatrixPointers function, every mesh container in the hierarchy will contain an array of bone offset matrices and a matching array of bone matrix pointers with a 1:1 correspondence between the two arrays.

Before we cover the CActor::BuildBoneMatrixPointers function, let us have a quick look at modifications to CActor::LoadActorFromX so that you can see where we have added this new function call. The code to this function will not be explained since we have now it many times before. We have highlighted in bold the new code that has been added.

```
HRESULT CActor::LoadActorFromX( LPCTSTR FileName, ULONG Options,
                                LPDIRECT3DDEVICE9 pD3DDevice,
                                bool bApplyCustomSets /* = true */ )
{
    HRESULT hRet;
    CAllocateHierarchy Allocator( this );

    // Release previous data!
    Release();

    // Store the D3D Device here
    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();
    m_nOptions = Options;

    // Load the mesh heirarchy and the animation data etc.
           D3DXLoadMeshHierarchyFromX( FileName,
                                       Options,
                                       pD3DDevice,
                                       &Allocator,
                                       NULL,
                                       &m_pFrameRoot,
```

```
                                        &m_pAnimController );

    // Build the bone matrix tables for all skinned meshes stored here
    if ( m_pFrameRoot )
    {
        BuildBoneMatrixPointers( m_pFrameRoot );
    }

    // Copy the filename over
    _tcscpy( m_strActorName, FileName );

    // Apply our derived animation sets
    if ( m_pAnimController )
    {
        // Apply our default limits if they were set before we loaded
        SetActorLimits( );

        // If there is a callback key function registered, collect the callbacks
        if ( m_CallBack[CALLBACK_CALLBACKKEYS].pFunction != NULL )
            ApplyCallbacks();

        // Apply the custom animation set classes if we are requested to do so.
        if ( bApplyCustomSets == true )
        {
            ApplyCustomSets();
        }
    } // End if any animation data

    // Success!!
    return D3D_OK;
}
```

CActor::BuildBoneMatrixPointers

This is a recursive function that is called once by the CActor::LoadActorFromX function, where it is passed a pointer to the root frame. The function will then recurse until it has visited every frame in the hiearchy. It is only interested in finding frames that have mesh containers that store skinned meshes. Once a skin is found, it will collect the bone matrices that it uses and store them in the mesh container's bone pointer array.

The first section of the function tests to see if the frame we are currently processing contains a mesh container that stores a skinned mesh (i.e., the pSkinInfo member will is non-NULL).

```
HRESULT CActor::BuildBoneMatrixPointers( LPD3DXFRAME pFrame )
{
    HRESULT hRet;

    // Has a mesh container?
    if ( pFrame->pMeshContainer )
    {
        D3DXMESHCONTAINER_DERIVED * pContainer =
                            (D3DXMESHCONTAINER_DERIVED*)pFrame->pMeshContainer;
```

```
        LPD3DXSKININFO pSkinInfo  = pContainer->pSkinInfo;

        // if there is a skinmesh, then setup the bone matrices
        if ( pSkinInfo != NULL )
        {
```

At this point we know there is a skin stored at this frame so we need to get the addresses of all bone matrices it needs to use so we can store them in the mesh container's array. This will provide us easy access to them during rendering.

The first thing we must do is allocate the mesh container's bone matrix pointer array so that there are enough elements to hold a matrix pointer for every bone it uses. The ID3DXSkinInfo for this mesh will tell us that information via its GetNumBones member.

```
        ULONG BoneCount = pSkinInfo->GetNumBones(), i;

        // Allocate space for the bone matrix pointers
        pContainer->ppBoneMatrices = new D3DXMATRIX*[BoneCount];
        if ( pContainer->ppBoneMatrices == NULL ) return E_OUTOFMEMORY;
```

We are now ready to start searching for bone matrices. The important point however is that we wish to find and add them to the array in the right order so that we have a perfect pairing between the bone offset matrix array and the bone matrix pointer array. This is not a problem because the ID3DXSkinInfo object contains every bone offset matrix and the name of the bone/frame it belongs to. Since we simply copied over the bone offset matrices into the mesh container (in CreateMeshContainer) in the same order, we can take advantage of this fact. We just have to loop through each bone in the ID3DXSkinInfo object and retrieve its name. We then search for a frame in the hierarchy with the same name and store the address of its absolute matrix in the array.

```
        // Store the bone pointers
        for ( i = 0; i < BoneCount; ++i )
        {
               // Find the matching frame for this bone
               D3DXFRAME_MATRIX * pMtxFrame =
                 (D3DXFRAME_MATRIX*)D3DXFrameFind( m_pFrameRoot,
                                                   pSkinInfo->GetBoneName(i) );
               if ( pFrame == NULL ) return E_FAIL;

               // Store the matrix pointer
               pContainer->ppBoneMatrices[ i ] = &pMtxFrame->mtxCombined;

        } // Next Bone
```

The above code does all the work. We set up a loop to iterate through each bone in the ID3DXSkinInfo and use the D3DXFrameFind function to perform a search (starting from the root) for a frame with the same name as the bone name. This function will return a pointer of that frame and we can copy its absolute matrix address into the bone pointer array. When this loop ends, all bone matrices for this mesh will have their pointers stored in the mesh container.

Usually this would be the end of the road, but we do need to perform one additional test to cater for the pure software skinning case. You will recall that when performing software skinning, we have to manually combine the bone matrices and the bone offset matrices into a temporary array before handing this combined array off to the ID3DXSkinInfo::UpdateSkinnedMesh function. Since all software skins only need to use this buffer temporarily during the update of their skin, we can create one matrix buffer that can be used by all software skinned meshes. Therefore, we have added a new matrix pointer m_pSWMatrices to our CActor class that can be used as a temporary matrix mixing buffer. We have also added a member called m_nMaxSWMatrices which will store the current size of this array. Since this array will only be used for software skins, it will be initially set to NULL and the m_nMaxSWMatrices variable set to zero.

This buffer must be large enough to cater for all the matrices used by the software skin with the maximum number of bones. We can use this same buffer for skins that have fewer bone combinations as this simply means that some of the elements in this array will not be used.

In the next section of code we test to see if the current skin being processed is a software skin. If it is, then we test to see if the current size of this temporary matrix array is large enough to facilitate this mesh. If not, then the temporary matrix array is resized to the number of bones used by the mesh. Obviously, the first time a software skin in encountered, this will always be the case because the array size will be set to zero. However, when processing other mesh containers later, we mind find other software skinned meshes that use even more bones and the array will be resized again. By the time the BuildBoneMatrixPointers function returns back to LoadActorFromX, this temporary matrix buffer will be large enough to handle any software skin in the hierarchy.

```
            // If we are in software skinning mode, we need to allocate our
            // temporary storage. If there is not enough room, grow our temporary
            // array.
            if ( pContainer->SkinMethod == SKINMETHOD_SOFTWARE
                 && m_nMaxSWMatrices < BoneCount )
            {
                // Release previous memory.
                if ( m_pSWMatrices ) delete []m_pSWMatrices;
                m_pSWMatrices = NULL;

                // Allocate new memory
                m_pSWMatrices = new D3DXMATRIX[ BoneCount ];
                m_nMaxSWMatrices = BoneCount;

                // Success ?
                if ( !m_pSWMatrices ) return E_OUTOFMEMORY;

            } // End if grow SW storage

        } // End if skinned mesh

    } // End if has mesh container
```

At this point we have fully populated the mesh container and there is nothing left to do at this frame. So, using the recursive behavior we have seen so many times before, we continue to walk the hierarchy, first visiting the sibling list and finally visiting the child list.

```
    // Has a sibling frame?
    if (pFrame->pFrameSibling != NULL)
    {
        hRet = BuildBoneMatrixPointers( pFrame->pFrameSibling );
        if ( FAILED(hRet) ) return hRet;

    } // End if has sibling

    // Has a child frame?
    if (pFrame->pFrameFirstChild != NULL)
    {
        hRet = BuildBoneMatrixPointers( pFrame->pFrameFirstChild );
        if ( FAILED(hRet) ) return hRet;

    } // End if has child

    // Success!!
    return D3D_OK;
}
```

At the end of this function, the entire hierarchy is complete. Our actor's mesh containers will have all the information they need in order to be transformed and rendered. The only new part of CActor that is left for us to look at is the rendering code.

CActor::DrawMeshContainer

Although our actor is now much more complex than it used to be, nothing has changed from the perspective of the application regarding how the actor is rendered. As in all of our previous lab projects that have used CActor, the scene is managing the textures and materials, so rendering each actor consists solely of looping through each subset and calling the DrawSubset method with the global ID for the subset we wish to draw.

You will hopefully recall that CActor::DrawSubset is a simple wrapper function around the first call to the recursive function CActor::DrawFrame. It passes in the root frame and the subset ID that was requested to be drawn by the application. The CActor::DrawFrame function then recursively walks the hierarchy looking for mesh containers. Once a mesh container is found, the CActor::DrawFrame function calls the CActor::DrawMeshContainer function, passing in the mesh container and the subset ID that needs to be rendered. It is in the CActor::DrawMeshContainer function that all the rendering work is done. This function has changed significantly from previous lab projects since it has been expanded to render both regular meshes and skins (indexed, non-indexed skins, and pure software). This function now also manages the setting of the bone matrices on the device (pipeline skinning) before the requested subset is rendered. In the software skinning case, the function updates the destination mesh in software if the hierarchy has been updated since the last frame.

DrawMeshContainer is divided up into four sections. Each section takes case of rendering a different mesh type. The first section handles the rendering of the mesh subset if it is an indexed skin. The second section contains the rendering logic for a non-indexed skin. The third section is renders a pure software skin, and the final section handles regular meshes. This is now a pretty large function (although not too complex), so we will cover it a piece at a time.

We will begin with the indexed skinning case. You can see below that we first test to see if the mesh container we are about to render contains a skinned mesh or not. If the mesh container's pSkinInfo member is set to NULL then we will skip the entire skinned code block and proceed to the bottom of the function where the regular mesh will be rendered normally. If a skin does exist, we first test to see if the skinning method is indexed.

```
void CActor::DrawMeshContainer(LPD3DXMESHCONTAINER pMeshContainer,
                               LPD3DXFRAME pFrame,
                               long AttributeID /* = -1 */ )
{
    ULONG                        i, j, MatrixIndex;
    D3DXFRAME_MATRIX           * pMtxFrame  = (D3DXFRAME_MATRIX*)pFrame;
    D3DXMESHCONTAINER_DERIVED * pContainer(D3DXMESHCONTAINER_DERIVED*)pMeshContainer;
    CTriMesh                   * pMesh      = pContainer->pMesh;
    LPD3DXSKININFO               pSkinInfo  = pContainer->pSkinInfo;
    D3DXMATRIX                   mtxEntry;

    // Is this a skinned mesh ?
    if ( pSkinInfo != NULL)
    {
        if ( pContainer->SkinMethod  == SKINMETHOD_INDEXED )
        {
            // We have to render in sofware
            if ( pContainer->SoftwareVP )m_pD3DDevice->SetSoftwareVertexProcessing( TRUE );

            // Set the number of blending indices to be used
            if ( pContainer->InfluenceCount == 1 )
             m_pD3DDevice->SetRenderState( D3DRS_VERTEXBLEND,D3DVBF_0WEIGHTS );
            else
             m_pD3DDevice->SetRenderState(D3DRS_VERTEXBLEND,pContainer->InfluenceCount-1);


            // Enable indexed blending
           if( pContainer->InfluenceCount )
              m_pD3DDevice->SetRenderState( D3DRS_INDEXEDVERTEXBLENDENABLE, TRUE );
```

The first thing we do in the indexed skinning case is test to see if the mesh container's SoftwareVP boolean is set to true. If it is, then it means we determined during the loading phase that this mesh has one or more triangles that are influenced by more bones than the device has matrix slots in its palette. If so, this mesh must be rendered using software vertex processing and we call the appropriate API call to make that so.

Our next task in the above code is to correctly switch on vertex blending in the pipeline using the D3DRS_VERTEXBLEND render state. We must set it for the value that describes the vertex format of the mesh. This assures that the pipeline knows the location of the indices within each vertex structure. The mesh container also stores the influence count which tells us the maximum number of bone influences per-vertex. This also tells us (if you subtract 1) the number of weights contained in the vertex structure. Remember that the last weight is calculated on the fly by the pipeline, so a three weight vertex structure would actually use four matrices. So, you can see that we first test to see if the influence count is equal to 1. If so, then every vertex in this mesh is influenced by only one bone. When this is the case, no actual vertex blending occurs since vertex blending assumes the influence of multiple matrices on a single vertex. Thus, the vertex structure will have no weights defined at all because the pipeline will assume a weight of 1.0 for the only matrix used for transformation.

However, if the influence count is not 1, then we know that by subtracting one from this amount, we have the actual number of weights stored in the vertices. This is exactly the value we should set the vertex blend render state to. It is very important that we do this correctly in the indexed skinning case because if we pass in a vertex blend renderstate of *n*, the pipeline will expect the indices to be stored in weight *n* + 1. If we set this value incorrectly, we are giving bad information to the pipeline about the format of our vertices, and it may try to fetch the matrix palette indices from the wrong weight causing all manner of incorrect transformations.

Finally, the last thing we do in the above code is test to see if the maximum vertex influence count is greater than 1. If so, we enable indexed vertex blending. This is another important point to remember. It is possible to render an indexed skinned mesh without using any form of vertex blending. This is typically the case if every vertex in the mesh is only influenced by a single bone. For example, we could have a skin that collectively uses a palette of 200 bones but no one vertex ever uses any more than one of those bones for its transformation process. In this case, while we are still rendering a skin using indices into the matrix palette for the mesh as a whole, no vertex blending needs to be performed for its vertex transformations. After all, if a vertex is only to be transformed by a single matrix there is nothing to blend. In such a case we simply transform the vertex by the single bone matrix. This is exactly why such a mesh would also have no need for weights in its vertices. So, we only enable indexed vertex blending when necessary.

Now that we have prepared the device to render our skin, our next task is to loop through each subset and render it. The information for each subset is stored in the mesh container's bone combination table, so we fetch a pointer to that first. We then loop through each subset/bone combination structure and process it. Take a look at the following code that renders each subset and we will talk about it afterwards:

```
            // Get a usable pointer to the combination table
            LPD3DXBONECOMBINATION pBoneComb = (LPD3DXBONECOMBINATION)
                                   pContainer->pBoneCombination->GetBufferPointer();

            // Loop through each of the attribute groups and calculate the
            // matrices we need.
            for ( i = 0; i < pContainer->AttribGroupCount; ++i )
            {
             if ( AttributeID >= 0 && pBoneComb[i].AttribId !=(ULONG)AttributeID )
                    continue;

                // First calculate the world matrices
                for ( j = 0; j < pContainer->PaletteEntryCount; ++j )
                {
                  // Get the index
                  MatrixIndex = pBoneComb[i].BoneId[j];

                  // If it's valid, set this index entry
                  if ( MatrixIndex != UINT_MAX )
                  {
                      // Generate the final matrix
                      D3DXMatrixMultiply( &mtxEntry,
                                          &pContainer->pBoneOffset[ MatrixIndex ],
                                           pContainer->ppBoneMatrices[ MatrixIndex ] );

                        // Set it to the device
                        m_pD3DDevice->SetTransform( D3DTS_WORLDMATRIX( j ), &mtxEntry );
```

```
                } // End if valid matrix entry

            } // Next Palette Entry

            // Draw the mesh subset
            pMesh->DrawSubset( i, pBoneComb[i].AttribId );

        } // Next attribute group
```

The first thing we do when we enter the subset loop is test to see if this subset is the subset that we actually want to render. Although we are looping through each subset, if the caller has requested that only subset *n* is to be rendered, we will skip any others. If no subset ID has been passed then this usually means that we are using managed meshes. When this is the case we wish to render every subset in the mesh since the CTriMesh will take care of setting the texture and material on the device before it renders the triangles. You can see in the above code that if the passed attribute ID is less than zero, no subsets wil be skipped and all will be rendered. This is the value that gets passed to the DrawMeshContainer function when managed meshes are being rendered.

Once we have decided that we wish to process a subset, our next task is to loop through each element in the BoneID array of the subset (which will have PaletteEntryCount elements). We do this to fetch the indices of the bones that influence the subset which need to be sent to the device prior to rendering. If any of the BoneId elements for a given subset are set to UINT_MAX then this subset does not use that matrix slot and the element can be ignored. The index of the BoneId element in the array describes the matrix slot that the bone index it contains should be set to. For example, if BoneId[5]=10, it means bone 10 in the mesh container's bone matrix array should be assigned to matrix slot 5 on the device. You can see that we use the bone index stored in each BoneId element to fetch the correct bone matrix and its corresponding bone offset matrix from the mesh container's array. We then multiply them together to create a single combined matrix which is set in its appropriate position in the matrix palette.

After we have looped though every element in the BoneId array for the current subset we are processing, we will have assigned all the bone matrices to their correct places in the device's matric palette and we are ready to render the mesh subset. We do this with a call to CTriMesh::DrawSubset. Notice that we pass in two parameters. The first parameter is the local subset of the mesh we wish to render. (Remember, we never re-map subset IDs when using skins.) The second parameter, which is used only in managed mode, describes the index for the texture and material used by this subset in the mesh's internal attribute table. Thus, the mesh can correctly set the texture and material on the device before rendering the subset. Obviously, if the mesh is in non-managed mode, this second parameter is unnecessary since the mesh assumes that the application has set the texture and material before making the draw call. If this is confusing, just remember that earlier in the lesson, we discussed how the AttribId member of a bone combination structure contains the original subset ID that this new subset was created from when the mesh was converted to a skin. This describes the index into the mesh's attribute data array for the texture and material used by that subset.

At this point, we have rendered our indexed skinned mesh (or a requested subset) and our job is done. Before we exit however, we should reset those device states we changed and set them back to their defaults. As you can see, in the closing statements of the indexed skinning case, we disable indexed vertex blending, set the vertex blending render state back to zero and return software vertex processing back to its default state.

```
            // Reset the blending states
            m_pD3DDevice->SetRenderState( D3DRS_INDEXEDVERTEXBLENDENABLE, FALSE );
            m_pD3DDevice->SetRenderState( D3DRS_VERTEXBLEND, 0 );

            // Reset back to hardware vertex processing if required
            if ( pContainer->SoftwareVP )
                 m_pD3DDevice->SetSoftwareVertexProcessing( FALSE );

        } // End if indexed skinning
```

In the next section we will look at the code that handles the non-indexed case. This code is very similar except for a few minor changes here and there. As before, the code starts by getting the mesh's bone combination table so that it has access to the bone and attribute information for each subset. We then loop through each subset to process and eventually render it as we did before. Once again, in the following code you can see that if a specific subset ID was passed in and the current subset we are processing does not match, we skip it. This ensures only the requested subset is rendered for non-managed mode meshes.

```
        else if ( pContainer->SkinMethod == SKINMETHOD_NONINDEXED )
        {
            // Get a usable pointer to the combination table
            LPD3DXBONECOMBINATION pBoneComb =
              (LPD3DXBONECOMBINATION)pContainer->pBoneCombination->GetBufferPointer();
            // process each subset
            for ( i = 0; i < pContainer->AttribGroupCount; ++i )
            {
                // Retrieve the remapped index
                ULONG iAttribID = pContainer->pVPRemap[ i ];

                // Enable software processing if we reached our SW begin index
                if ( i == pContainer->SWRemapBegin )
                    m_pD3DDevice->SetSoftwareVertexProcessing( TRUE );

                if ( AttributeID >= 0 &&
                     pBoneComb[ iAttribID ].AttribId != (ULONG)AttributeID ) continue;
```

At the center of the subset loop there is a fundamental change. With indexed skinning, the mesh is either in software or hardware mode, but with non-indexed skinning there may be a mixture of both types. You will recall that to speed up the switching process, we created a subset re-map array so that the subsets are stored and processed in order of hardware capability (hardware capable subset IDs followed by software subset IDs). So we will process the subsets not in the usual 0 to *n* order, but in the order they are stored in this re-map array. The code above does just that.

We fetch the bone combination table and then set up a loop to iterate through the number of subsets. At the start of each iteration we fetch the next subset we wish to process from the re-map array. We then test the current loop variable 'i' to see if it is equal to our mesh container's SWRemapBegin. We calculated this value earlier as the position in the re-map array where hardware subsets end and software subsets begin. If we hit that index, then it is time to enable software vertex processing for the rest of the subsets we process. Finally, we test the subset ID against the subset ID passed into the function (AttribID). If this is not the subset our application specifically wished to render then we skip to the next iteration of the loop.

Now we know which subset we are processing and we also know which bone combination structure to use, so we can fetch the bone matrix indices it uses. We set up a loop to loop through each element in the BoneID array. This will be a maximum of four in the non-indexed case, but may be less if the hardware does not support all four matrix slots. This value will be stored in our mesh container's InfluenceCount member as in the indexed case. In the non-indexed case, this value actually contains the total number of bones that influence a given subset, but is pretty much the same thing as far as determining the size of the palette being used is concerned.

As discussed earlier, in the non-indexed case we do not have matrix indices in the vertices to worry about, so we can set the D3DRS_VERTEXBLEND render state on a per-subset level so that we do not unnecessarily blend using matrices that have zero weights defined in the vertex. However, in order to perform this optimization, we must know how many matrices this subset uses (among all its vertices). In the following loop, we simply test each BoneId element for this subset to see if a valid BoneId is stored there. Each time we find one, we set the local variable MaxBoneIndex equal to loop variable 'j'. At the end of this loop, MaxBoneIndex will contain a value between 0 and 3 describing the highest matrix index that influences this subset. If this was 2 for example, then we know that we can set the D3DRS_VERTEXBLEND render state to process only 1 weight (the second weight is calculated by the pipeline). This would save us having to incorporate matrices 2 and 3 into the calculation unnecessarily as their contribution would have zero weight anyway. Remember, without doing this step, if one vertex in the mesh was influenced by four bones, but all the rest were influenced by only one, we would multi-matrix transform every vertex in the mesh using four matrices, even if those matrices contribute nothing.

```
                ULONG MaxBoneIndex = UINT_MAX;

                // Count the number of bones required for this attribute group
                for ( j = 0; j < pContainer->InfluenceCount; ++j )
                {
                    // If this is not an 'empty' bone, increase our max bone index
                    if ( pBoneComb[ iAttribID ].BoneId[ j ] != UINT_MAX ) MaxBoneIndex = j;

                } // Next Influence Item
```

We now know how many matrix slots this subset uses. So let us take another pass through the BoneId array and actually build the matrices this time. For each BoneId element in the array, we fetch the Bone index it describes. We then use this index to fetch the bone matrix and the bone offset matrix for the mesh container's arrays and bind them to the device in the correct matrix slot.

```
                // First calculate the world matrices
                for ( j = 0; j <= MaxBoneIndex; ++j )
                {
                    // Get the index
                    MatrixIndex = pBoneComb[ iAttribID ].BoneId[j];

                    // If it's valid, set this index entry
                    if ( MatrixIndex != UINT_MAX )
                    {
                        // Generate the final matrix
                        D3DXMatrixMultiply( &mtxEntry,
                                            &pContainer->pBoneOffset[ MatrixIndex ],
                                            pContainer->ppBoneMatrices[ MatrixIndex ] );

                        // Set it to the device
```

```
                        m_pD3DDevice->SetTransform( D3DTS_WORLDMATRIX( j ), &mtxEntry );

                    } // End if valid matrix entry

                } // Next Palette Entry
```

At this point the device has been assigned all the matrices needed to transform and render this subset. We will now set the vertex blend render state to process MaxBoneIndex weights and finally draw the actual subset using CTriMesh::DrawSubset.

```
                // Set the blending count to however many are required / in use
                m_pD3DDevice->SetRenderState( D3DRS_VERTEXBLEND, MaxBoneIndex );

                // Draw the mesh subset
                pMesh->DrawSubset( iAttribID, pBoneComb[ iAttribID ].AttribId );

            } // Next attribute group
```

With every subset of the mesh now processed and rendered, all that is left to do before exiting the non-indexed skinning case is reset the render states we may have changed to their default settings.

```
            // Reset the blending states
            m_pD3DDevice->SetRenderState( D3DRS_VERTEXBLEND, 0 );

            // Disable software processing if it was enabled
            if ( pContainer->SWRemapBegin < pContainer->AttribGroupCount )
                 m_pD3DDevice->SetSoftwareVertexProcessing( FALSE );

        } // End if Non-Indexed
```

We have now covered the rendering section that deals with both pipeline skinning methods. The final skinning method we have to deal with is pure software skinning. In many ways, software skinning is much simpler since we do not have the bone combination relationship to deal with and we do not have to worry about tweaking the vertex blend and software vertex processing render states.

In software skinning the basic process is:

1) Combine every bone matrix with every bone offset matrix used by the mesh and store in our temporary array.

2) Lock the vertex buffer's of the model space source mesh and the destination mesh.

3) Pass the combined matrices and both vertex buffers into the ID3DXSkinInfo::UpdateSkinnedMesh function. On function return, the destination mesh will contain world space geometry that has been calculated using multi-matrix transformations.

4) Set the device world matrix to an identity matrix (destination mesh already in world space)

5) Render the mesh

The basic process is not new to us but is listed above for your review. There are one or two things in the code that we have added for efficiency and convenience which will be discussed before we look at the code.

First, we really do not want to regenerate the skinned mesh (as described above) every time we wish to render it. If none of its bones have changed position since the last build then this is unnecessary since the world space geometry in the destination mesh will still be current. As mentioned earlier, we have added a boolean variable called Invalidated to our mesh container structure. The mesh only gets rebuilt if this value is set to true; otherwise we simply render the current destination mesh. The Invalidated boolean is set to true (for every mesh container) when the hierarchy is updated (e.g., after animation has been applied) inside the CActor::UpdateFrameMatrices function. If it is set to true when it comes time to render this mesh, we will rebuild the world space destination mesh (prior to rendering it) and then reset this Boolean back to false after this has been done. This way, we only rebuild the destination mesh when a change occurs to the hierarchy.

Below you will see the code to all the steps outlined above. It fetches every bone matrix and bone offset matrix stored inside the mesh container and combines each pair into a resulting temporary matrix buffer (m_bSoftwareMatrices). It then locks the vertex buffers of the source and destination mesh prior to passing them into the ID3DXSkinInfo::UpdateSkinnedMesh method along with the combined matrix buffer. This is the code for the software skinning case:

```
        else if ( pContainer->SkinMethod == SKINMETHOD_SOFTWARE )
        {
            D3DXMATRIX  Identity;
            ULONG       BoneCount = pSkinInfo->GetNumBones();
            PBYTE       pbVerticesSrc;
            PBYTE       pbVerticesDest;

            // Get the actual mesh
            LPD3DXBASEMESH pSrcMesh = pMesh->GetMesh();
            if ( !pSrcMesh ) pMesh->GetPMesh();

            // If the data has been invalidated since last rendering, update our SW mesh
            if ( pContainer->Invalidated )
            {
                // Loop through and setup the bone matrices
                for ( i = 0; i < BoneCount; ++i )
                {
                    // Concatenate matrices into our temporary storage
                    D3DXMatrixMultiply( &m_pSWMatrices[i],
                                        &pContainer->pBoneOffset[i],
                                        pContainer->ppBoneMatrices[i] );

                } // Next Bone

                // Lock the input and output vertex buffers
                pSrcMesh->LockVertexBuffer(D3DLOCK_READONLY, (LPVOID*)&pbVerticesSrc);
                pContainer->pSWMesh->LockVertexBuffer( 0, (LPVOID*)&pbVerticesDest);

                // Generate the newly skinned mesh data
                pSkinInfo->UpdateSkinnedMesh( m_pSWMatrices,
                                              NULL,
                                               pbVerticesSrc,
                                               pbVerticesDest );
```

```
                // Unlock the buffers
                pSrcMesh->UnlockVertexBuffer();
                pContainer->pSWMesh->UnlockVertexBuffer();

            } // End if update required
```

At this point, our destination ID3DXMesh (pSWMesh) contains the new position of our software skin's vertices in world space. Therefore, we must make sure when we render it that the world matrix is set to an identity matrix since we do not wish an additional world transform to be applied.

```
            // Already in world space so set world to identity
            D3DXMatrixIdentity( &Identity );
            m_pD3DDevice->SetTransform( D3DTS_WORLD, &Identity );
```

The next section of code requires some explanation. The world space vertices are currently stored in pSWMesh. This is just an ID3DXMesh and, as such, has no attribute data linked to it. We cannot use the CTriMesh interface to render this mesh because it is not attached to a CTriMesh. The CTriMesh stored in our mesh container is currently attached to the source mesh (pSrcMesh) which contains the model space skin data. Therefore, we will temporarily attach the destination mesh to the CTriMesh instead, so that we can use its draw functions and borrow is texture and material data (for managed mode meshes).

```
            // Attach the SW mesh to our CTriMesh (attach only) for rendering
            pMesh->Attach( pContainer->pSWMesh, NULL, true );
```

We are now ready to render the CTriMesh since it currently contains the world space mesh geometry. If the application specified a subset to be rendered, then we call CTriMesh::DrawSubset. If not, we assume this is a managed mode skin and call CTriMesh::Draw. We know from previous lab projects that this function simply calls DrawSubset for each subset in the mesh.

```
            // Render the mesh or subset
            if ( AttributeID >= 0 )
            {
                // Draw the subset
                pMesh->DrawSubset( (ULONG)AttributeID );

            } // End if attribute specified
            else
            {
                // Draw the mesh as a whole
                pMesh->Draw( );

            } // End if no attribute specified
```

With the mesh now rendered, we re-attach the CTriMesh's original ID3DXMesh which contains the geometry in its model space reference pose. We do this to be consistent for the application. In all other skinning techniques and in the case of regular meshes, the application could access the CTriMesh stored at each mesh container and would expect it to be in model space. By switching it back again, we make sure that this assumption is correct when software skinning is being used. After all, we never know what somebody will want to do with our actor. For example, they may be traversing our hierarchy in order to build model space bounding volumes for each of the meshes.

```
            // Re-attach our source mesh
            pMesh->Attach( pSrcMesh, NULL, true );

            // Now we can release our source mesh
            pSrcMesh->Release();

        } // End if software skinned
    } // End if skinned
```

That ends the software skinning conditional code block, and consequently, the entire skinned mesh rendering code block. The only conditional code block left to cover is executed if the mesh stored here is not a skin, but a standard mesh. When this is the case, we render the mesh as we always have.

```
    else
    {
        // Render the mesh
        if ( AttributeID >= 0 )
        {
            // Draw the subset
            pMesh->DrawSubset( (ULONG)AttributeID );

        } // End if attribute specified
        else
        {
            pMesh->Draw( );
        } // End if no attribute specified
    } // End if not skinned
}
```

While that was quite a big rendering, each rendering method taken individually is fairly small.

Conclusion

We have now covered all of the modifications to CActor that provide automatic support for the loading, transformation, and rendering of skinned meshes. There is no need to look at modified code in the CScene or CGameApp class because, there is none. We have managed to expand our actor functionality without the application needing to know about the changes. Thus, we have accomplished all of our design goals that we stated early in the lesson.

In the next lesson we will continue to explore what can be achieved with both animation and skinned meshes. In the workbook, we will add a data-driven animation layer to CActor, allowing us to select which combinations of animation sets we wish to play on the fly in response to use input or other game events. In our textbook, we will also gain a much deeper understanding of how the data for a skinned mesh is laid out when we create a new class derived from CActor that will be used to procedurally generate skinned tree meshes. The examination of the CTreeActor code will demonstrate how we can build skinned meshes and skeletal hierarchies procedurally as well as how to generate procedural animation for the skeletal structure. This should be a nice way to round off our current studies of skinned meshes and skeletal hierarchies. The combination of all of these concepts in our workbook will make for a good way to mark the halfway point in this course.

Chapter Twelve

Skinning II

Introduction

At this point in the course we have learned how to load X file frame hierarchies and traverse and render them. We have also learned how to load skinned mesh data from an X file and render those skinned meshes such that the hierarchy acts as a system of bones that influence the world space positions of the skin's vertices. Of course, loading data from an X file is only half the story. We can also procedurally generate meshes, hierarchies, bones, and skins. This can prove to be handy for lots of things, like random generation of terrain or even for vegetation on that terrain to cite just two obvious examples. Typically such code uses algorithms that generate geometry based on both fixed rules and some degree of randomness. How the two concepts work together is generally based on some set of input parameters and controls to the procedure. For example, if we were randomly generating a terrain mesh, we might specify some rules to the procedure stating that water can only exist below a certain altitude, or that snow can only be placed on mountaintops above a certain altitude. In both cases, those altitudes are likely to be input to the terrain generator via a configuration file or through a GUI tool.

Figure 12.1

Regardless of the example we choose, the important point is that you will undoubtedly reach a point at some time in your programming career where you will have to generate meshes and/or texturing information from within your code. For some game assets it is just easier to do it this way (versus getting your artists to create everything by hand). Trees are a good example. Every tree is different in reality, so if you wanted to represent even a small forest of say, 40 unique trees, it would require the artist to generate 40 different tree meshes, where each branch and maybe even each leaf would have to be carefully hand-crafted and textured. The time involved would be considerable, and most serious game development projects would simply not have the resources to spare for such an undertaking. So not surprisingly, representing realistic trees within a game world has typically been an area that has fallen to tools developers to aid the creative process. There are currently some commercial products (e.g., SpeedTree™) available that are designed to do nothing more than generate realistic trees that look organic while remaining within a specified polygon budget (the polygon count of a single tree is very important as we rarely have just one tree in a given scene – especially an outdoor scene). These products produce some incredibly realistic results, but they can also be very expensive. Since commercial tools are generally not in the budget for most students, we decided to introduce code for generating our own trees. This will be a great way to conclude our studies of skins and skeletons and it has the added benefit of providing you with a useful tool in the short term at no extra cost (other than the time you will have to spend writing the code). To be clear, we are not going to claim that our trees are going to exhibit the same level of quality or performance as those produced in the commercial tools, but we are sure that you will find them to be an adequate substitute in the short term while you are still in training.

In Figure 12.1 we see a tree that has been generated procedurally using a new class that we will create in this chapter (called CTreeActor). What cannot be seen in the image is that this tree is really a combination of skinned meshes and bones. The bones are animated to simulate how a tree might move in blowing wind. Our tree class will be derived from CActor and as such we can reuse much of its functionality. CActor already has code to represent, animate, and render skinned meshes, so this is code we will not have to rewrite in our tree class (thankfully). In fact, CTreeActor will simply add some hierarchy and skinned mesh creation functions to the base class that allow us to populate the actor's hierarchy using a different means. As we have seen, currently CActor can be populated only by loading an X file. With CTreeActor we will add functions that generate the bone and mesh data for tree meshes and add them to the hierarchy manually.

> **Note**: While it is perfectly legitimate to use CTreeActor to generate trees in your code at level load time (for example), it is really supposed to be used as tool. That is, usually you will use CTreeActor to generate some number of random trees and when it creates a tree you like, you will save that actor to an X file for use in your game. This way, your actual game code need only load the trees using a normal CActor just like any other X file. You may also wish to save the trees out to X files and then import them into a level editor for placement. GILES™ now ships with a tree plug-in to allow you to create trees from within its GUI. The GILES™ tree plug-in uses CTreeActor for its tree generation.

A CTreeActor object will represent a single tree in the scene. Its frame hierarchy will describe the bones of the entire tree representation (the skeleton of the trunk and all its branches). CTreeActor's creation functions will also generate animation data for this hierarchy programmatically that will animate the tree such that its branches and leaves blow from side to side in simulated wind. Sharing this same bone hierarchy will be multiple meshes, one skinned mesh for each branch. What we will have at the end of this chapter (and associated lab project) is a class that we can use to generate tree meshes with random configurations with a single function call. For the most part, using the tree class will be no more complicated than calling the CTreeActor::GenerateTree function where we would usually call the CActor::LoadActorFromX function. Therefore, the code that we have added to CTreeActor is strictly confined to the creation process as a replacement for loading of X file data. Once the CTreeActor::GenerateTree function has built the frame hierarchy and its various branch meshes, we can position, animate, and render the tree using the same CActor methods we have always used.

Since this is going to be a fairly complex task, we will break the coverage of CTreeActor into two separate lab projects (12.1 and 12.2) which will be covered in two different sections in this textbook. In the first section of this book we will write the code that generates the main branch structure of the tree (Lab Project 12.1). This will involve procedurally creating multiple skinned meshes (one per branch). Once we have our CTreeActor fully capable of creating such a network of animating skinned branch meshes, we will add simple leaves to our trees (Lab Project 12.2). Doing it in two steps like this will make the code easier to understand and the entire process a little less overwhelming.

> **Note:** This chapter obviously represents a departure from the way we have done things thus far in the training series. Generally, source code discussions were limited to our workbooks while the textbooks concentrated more on higher level theoretical ideas. So it is worth noting that the line between textbooks and workbooks is likely to blur quite a bit as we move forward in this course (and indeed in the remainder of the series). As we start to get into more implementation specific topics (such as we will here), it will become difficult and impractical to split things up as cleanly as we have done in the past. Instead what we will find is that the work will be shared between the two books. For example, there will be times when the textbook will cover the implementation details in addition to the design ideas and

theory, while the workbook focuses on how to integrate and use the systems developed in the textbook. To be sure, this will not happen all the time, and again, the line will be somewhat blurred. Certainly there will be times in the future when the old model shows itself again and the code will be confined almost exclusively to the workbooks. But when it is not practical to do so, we will approach the topics like you will see here in this chapter. Indeed, throughout the rest of this course, you should expect to see this approach being used in nearly every chapter.

Also note that since this chapter is essentially just an extension of the ideas introduced in the last chapter, there will be no new accompanying presentation. We are going to focus exclusively on writing code this time around. Of course, the material in Lecture 11 is completely applicable to what we will do here, so please refer to the presentation as the need arises.

Figure 12.2

The tree in Figure 12.1 essentially consists of two geometry building techniques. Skinned meshes are used to represent the trunk and branches of the tree, while the leaves are represented using a series of criss-crossed quads. As we will be discussing the technique used for leaf generation in a later section, in this first section we will focus on creating only the trunk and branches of the tree. In essence, our first version of CTreeActor will only generate "winter" trees. Figure 12.2 shows what the tree shown in Figure 12.1 will look like without the leaves. This is what the trees we generate in the first section of this chapter will look like.

Of course, while it is essential to have trees at your disposal if you ever intend to represent an outdoor environment other than the Sahara Desert or the Atlantic Ocean, providing you with organic looking trees was not the motivation behind this lesson. The real goal behind this project was to come up with a task that would put to the test everything you have learned about since the beginning of the course, but with some new twists. For example, you learned how keyframe data is generated back in Chapter Ten, but so far we have only demonstrated how to load and playback that data. Similarly, in Chapter Eleven, we discussed the relationship between vertices in a skin, the bones that influence them, and the individual animations that animate those bones in the hierarchy. But even that material was focused on working with pre-generated data. In short, to really understand a skeletal animation system (or arguably, any system for that matter), it will be helpful to work on a project where every bit of data is generated from scratch right in your own code. This is exactly the educational experience that CTreeActor will provide, as you can see just by glancing at our to-do list:

- We will need to generate the meshes for each branch and of course, the bones which those meshes attach to.

- We will have to calculate the bone offset matrix for each bone in our hierarchy such that it is relative to the bone at which the branch mesh begins and not the root of the entire tree.

- We will have to manually connect the vertices to the bones that animate them.

- We will have to create an animation controller for the actor, which we will then populate with keyframe data that we will generate.

It seems that we have quite a bit of work to do, and indeed we do. But believe it or not, the actual code we will write to extend CActor into our derived tree class will be fairly small. It is still quite a complex process that we will be undertaking however, which is why we will first discuss the system design and try to nail down how it will all work. Then we can start dealing with source code and implementation details.

12.1 An Overview of the Tree Generation System

Our tree will be made up of a number of branches that will be randomly generated using a recursive process. Even the trunk of the tree will just be a branch (considered the "root branch" of the tree). Each branch will be "grown" using a recursive and random process that may, at any given segment within the branch, cause another child branch (or multiple child branches) to be generated. Each child branch generated from the root branch (the trunk) will also be built in exactly the same way. As each child branch is grown, there is a chance that any given segment within that branch may spawn multiple child branches of its own, and so on. Do not worry too much about what a branch segment is at the moment, we will get to that in a moment.

We will feed the generator procedure a number of input parameters that will be used to influence the growth of the tree and its individual branches. For example, one parameter we will feed in will be a growth direction vector that will describe the world space direction in which we wish the tree (the trunk branch) to initially start growing. We will also feed in probability variables that describe the odds that a branch segment might split into one, two, three or four more child branches at any given segment along its length. In addition we will feed in variables that influence the amount that a child branch's direction vector can be randomly deviated from its parent branch (while still maintaining the general overall growth direction of the tree as a whole). Further, we provide variables that allow us to specify the size of the tree and the resolution at which we place bones within that tree. Other variables will specify the resolution of the mesh branches, allowing us to control exactly how many vertices are used to create a given branch segment. This will allow us to directly influence the face count of the tree and tailor the tree to our specific polygon budget. So before we discuss any code, let us first examine the system that we will use at a high level. This way when we discuss the code later on, we will understand the input variables and how we are using them.

Figure 12.3 : A branch made out of a single cylinder

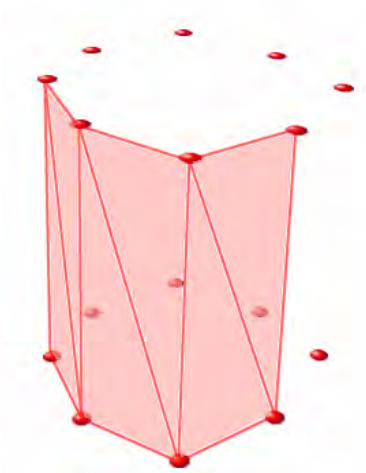


Figure 12.4: The two rows of vertices created triangles that wrap around a cylinder

As mentioned, our tree will essentially just be a hierarchy of branches. If we think in terms of primitive objects, a branch could be represented at a very coarse level by a cylinder mesh (see Figure 12.3). Already, this image looks like a very rough approximation of the trunk of a tree, especially when it is mapped using a tree bark texture. What we are looking at in Figure 12.3 is a single cylinder mesh. It has eight vertices arranged in a circle around the top and eight vertices arranged in a circle around the bottom of the cylinder. In this image we can only see four of the eight vertices used at the top and the bottom of the cylinder as we are only looking at one of its sides. As Figure 12.4 clearly demonstrates, we can use these two rows of vertices to create triangles that form the cylinder primitive. Each pair of vertices from the top and bottom rows can be indexed to form a quad (two triangles). When we do this for every matching pair of vertices in the top and bottom rows, we have an un-capped cylinder, which when textured, would look like the branch shown in Figure 12.3.

Having vertices only at the top and the bottom of each branch (one cylinder), does not afford us much flexibility when it comes to shaping the branch into something more interesting. While we could decrease the radius of the circle formed by the top row of vertices to make the cylinder get thinner as it nears its end, this would still be a perfectly uniform thinning out of the branch from bottom to top. We might imagine for example, that we would want the branch to end in a tip, possibly by moving all vertices in the top row into a single center point. However, the branch would then become a perfect cone and we really cannot have a tree where all the branches are perfectly coned shaped. So instead of making each branch using a single cylinder, a branch will be comprised of multiple cylinders stacked on top of each other.

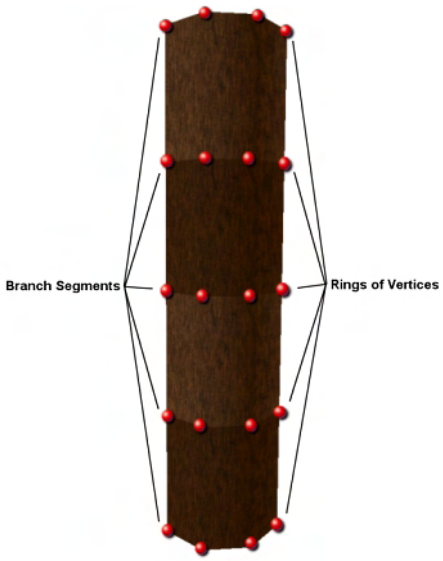


Figure 12.5 : A branch mesh constructed from multiple branch segments

We will refer to each cylinder as a **branch segment** as shown in Figure 12.5. Our branch will now consist of not just two rows of vertices arranged in a circular pattern, but *N* rows of vertices which form multiple branch segments. With the exception of the first row of vertices (at the bottom of the branch) and the last row of vertices (at the top of the branch), all other rows will consist of shared vertices that form the top of one branch segment and the bottom of another. In Figure 12.5, we show how our branch mesh might look consisting of multiple branch segments. All segments still belong to the same mesh, after all, they are all part of the same branch and we have already mentioned the fact that each branch will be its own skinned mesh.

Now that we have vertices at regular intervals along the branch, we can adjust the radius of the circles formed by each row of vertices such that the faces that form the branch get thinner and thicker as we desire. In our code, we will use the index of the branch segment within the branch we are currently building to determine the radius of its vertices (with some degree of randomness thrown in) such that all branches generally get thinner until they end in a point at their tip. It would be a waste representing the tip of any branch using a row of eight vertices that essentially exist in the same position, so when adding the final row of vertices to a branch (to complete the final segment of a branch), we will insert just a single vertex that exists at the center of the circle. Every vertex in the penultimate row of vertices will be indexed along with this final vertex to turn the final branch segment into a cone.

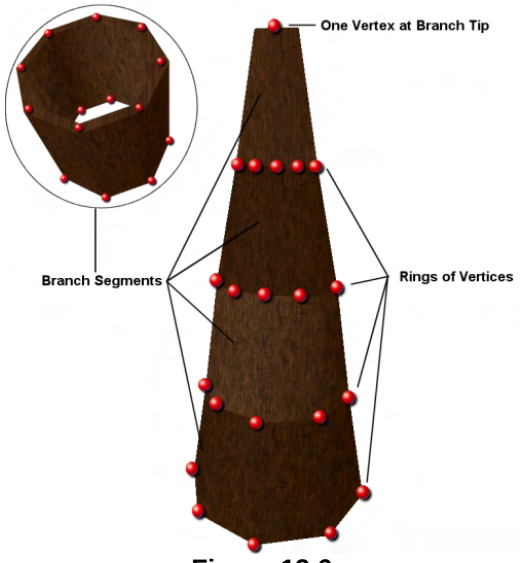


Figure 12.6

Figure 12.6 shows how a branch might look with uniform scaling applied to the vertices at each segment boundary. It also demonstrates how the final branch segment of any branch is terminated using a single vertex at its tip. While this shows a uniform scaling making the branch appear cone shaped, this is not something we are limited to doing. By changing the radius of the circle formed by each ring of vertices we can make the branch mesh swell or contract at the segment boundaries. The circular inset in figure

12.6 also reminds us that all branch segments (except the top segment) are essentially uncapped cylinders that slot together as a natural product of the fact that they share the same vertices at the segment boundaries. The top segment is obviously an exception to this rule where the faces of this segment will each be generated using the final 'tip' vertex and each pair of vertices from its base row of vertices.

So, each branch will be a mesh made up from a number of branch segments. Furthermore, each tree will be made from a number of branch meshes. In Figure 12.7 we see a tree that is made up of 7 branches. Admittedly it is a poor looking tree and there is much left to be done, but the relationship shown here is the important concept to understand.

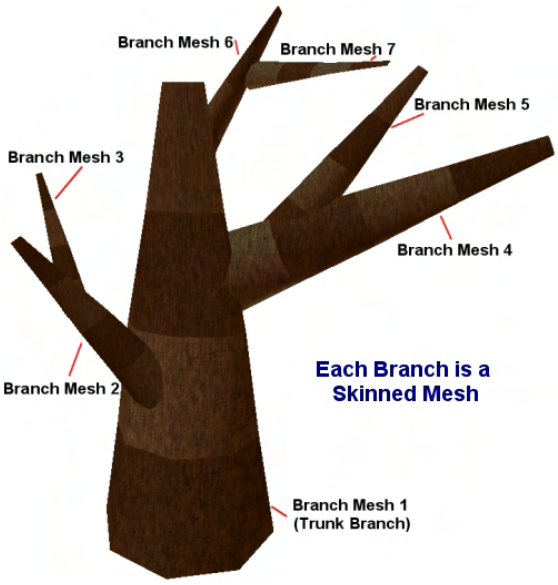


Figure 12.7

When we grow the tree, we initially start out with the root branch; more specifically, the root segment of the branch. We build the branch up segment by segment, at each step determining how big or wide that segment should be. We will also make a choice at each branch segment whether a new child branch (or multiple child branches) should be spawned from that segment. We can see in Figure 12.7, that at segment 2 of the trunk branch, a child branch was spawned off to the left, which itself set in motion a recursive procedure of building the child branch segment by segment in the same way. In this child branch we can see that while generating its third segment, another (smaller) child branch was spawned. After the recursive process ends for the child branches, the flow once again returns to the root branch, which is still half way through being built. On the third segment of the root branch it is determined again that another child branch should be spawned (this time off to the right), which itself spawns its own child branch during the construction of its second segment. When flow returns back to the root branch, and its fourth segment is added, the decision to split into a child branch is made yet again. This child also splits off into a separate child branch some way along its length.

We can definitely see the hierarchical pattern here. If we think about how we might represent this concept in a hierarchy (much like our frame hierarchy), in a way that is totally abstracted from any type of mesh data, we can think of the root segment of the trunk branch as being the root node in the hierarchy. The root node would have a child that would be the second segment in the root branch. This second level child would also have a child which would represent the third segment of the trunk branch, and the third segment would have a child that pointed at the data for the fourth and final branch segment in the trunk. So, each individual branch would be represented in the hierarchy as a list of N nodes arranged in a strict parent-child relationship that represented the N segments of that branch.

Now let us think about how the child branches could be stored. The first child node of the trunk branch in the hierarchy (which represents the second segment of the trunk) could have a sibling that is the first segment in the first child branch (the one that branches to the left). After all, the second segment of the trunk and the first segment of the child branch exist at the same level in the tree and should also exist at

the same level in the hierarchy. We then recur with that sibling pointer, such that each of its segments would be added in a parent-child relationship. So segment N of any branch is the parent of segment N+1 in the same branch.

In Figure 12.8, we see how this hierarchy might look in memory. To reduce clutter, we have only fully shown the trunk branch and the first child branch (protruding from the second segment of the trunk out to the left) in their entirety. When studying this image, do not concern yourself with meshes or bones for now; we are simply constructing a 'virtual tree' tree at this stage, where each branch segment has its own node in the hierarchy and the first segment in any child branch is connected as a sibling of the parent branch segment from which it was spawned.

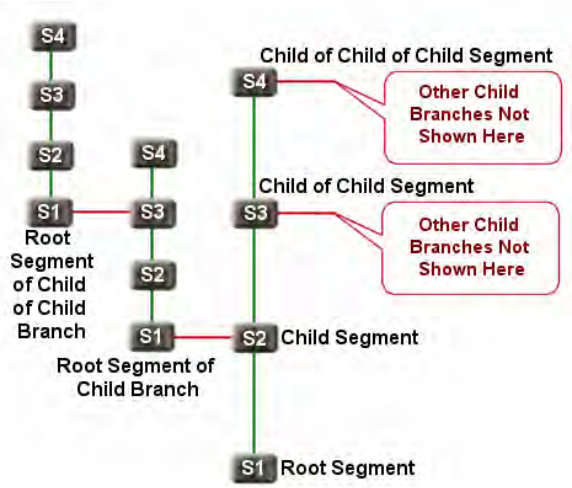


Figure 12.8

In Figure 12.8 we have inverted the diagram so that it reads bottom up. While this is not the typical way to draw a hierarchy, it corresponds better visually with the tree we saw in Figure 12.7. At the bottom of the image we see the root node of the hierarchy. This represents the first segment of the trunk branch. The parent/child relationship is depicted by vertical lines connecting the nodes, while the horizontal lines depict where a sibling relationship exists between two nodes. We can see that the root segment actually has two children arranged in a sibling list. The first child (S2) represents the second segment of the trunk branch, while a horizontal line connects node S2 to the root segment of a child branch. S2 of the root branch also has a child S3. S3 would also have a sibling but that is not shown here. The S3 node of the root branch has a child also (S4), which once again, would also have a sibling to the start of a child branch which is not shown here.

If we backtrack to node S2 of the root branch, we can see that it is in a sibling list with node S1 of a child branch. This tells us that at node S2 in the root branch, a new child branch was spawned. Node S1 in the child branch has one child (S2), but node S2 of the child branch has two children arranged in a sibling list. It points to node S3 (the third segment) in the child branch and node S1 in a new child that is spawned from this child branch at this third segment.

Study the diagram and make sure you understand the relationships as this will all be very important moving forward. If our tree was a single branch with four segments for example, this would create a four level deep hierarchy where each child (except the root) would be a child of the previous node. Each

level in the hierarchy would contain a single node. Any child branches that are spawned at a given branch segment, are arranged in a sibling list with that parent branch's segment node.

It is very important that you consider the node hierarchy shown in Figure 12.8 abstracted from the concept of meshes, vertices, or bones. We are currently just generating a hierarchy of data structures (nodes) that will describe to us the shape of the tree we need to generate. We might consider each node in the tree containing such information as the tree space position of that node and the direction it is facing. In fact, this is exactly how we will build our tree. We will essentially do it in two phases.

In the first phase, we will grow (node by node) a hierarchy of data structures describing the virtual tree. Then, once the shape of the virtual tree has been created, we will traverse this node hierarchy in a second phase and use the information to build the meshes and add the bones. This approach is preferable because it allows us to abstract the shape of the tree that we generate from the procedure used to skin it. After we have built the node hierarchy, we will traverse it in a second phase, and insert a ring of vertices at each node adding another segment to the branch. Of course, at a later time, you could decide to build the mesh data in a completely different way still using the same virtual tree information. For example, you might decide to use the node information to build the tree out of curved surfaces or even decide to drop the resolution such that a ring of vertices is only inserted every three nodes. These are just examples, but hopefully you understand why it is helpful to abstract the virtual tree generation process from the process that turns that virtual representation into a discrete polygonal representation.

As stated above, in our code, when we build the mesh, we will insert a ring of vertices at every node we generated in the virtual tree creation process. This means, with the exception of the first node of every branch, every other node we encounter will cause another branch segment (cylinder) to be generated. Obviously, the first node in a branch cannot possibly add another segment by itself as we need two rings of vertices to create a cylinder. Therefore, it is not until the second node is encountered and another ring of vertices is inserted that the first segment of the branch is complete. The ring of vertices we add at the second node however, also forms the bottom row of vertices for the next branch segment. So when we encounter the third node and add a third ring of vertices to the branch, this (along with the vertices from the second node) forms the second branch segment of the current branch being built.

Thus, our recursive procedure will initially grow a virtual tree out of a hierarchy of nodes, where each node contains data about that node. What information would we need to store in each node? First we will need to store the position of the node in tree space. In tree space, the root node of the trunk branch would be positioned at (0, 0, 0) in the coordinate system. However, what might not seem obvious at first is that in order to grow this node and spawn child nodes, we will need each node to also have a direction vector. This direction vector is a vector pointing in the direction of where the next node will be placed. More importantly, the direction vector can be thought of as a normal to a plane that passes through that node. When we generate the actual mesh for the branch, this is the plane upon which we will place the ring of vertices generated by that node.

To better understand the need to store the direction of a node, let us first just consider a four segment branch that forms the trunk of a tree growing directly upwards. In this case, the direction vector of each branch would simply point straight up to the next node (0,1,0). When we generate the mesh for the vertical tree, this direction vector also describes the normal to the plane on which the node is sitting. In this instance, each node would sit on an XZ aligned plane offset some distance vertically from the origin

of the coordinate system (except the first node which would exist on the XZ plane). As Figure 12.9 demonstrates, when we generate the mesh for a branch, the direction vector describes the normal of a plane. The node is assumed to be positioned at the center of a circle defined on that plane. Using the plane normal we can also generate two tangent vectors (vectors that lay on the plane forming an orthogonal axis with the normal vector). Essentially, once we have the three axes of the node, we can generate the N vertices at the node position and then push them out along the tangent vectors by some radius to form a circle of vertices that lay on the node's plane.

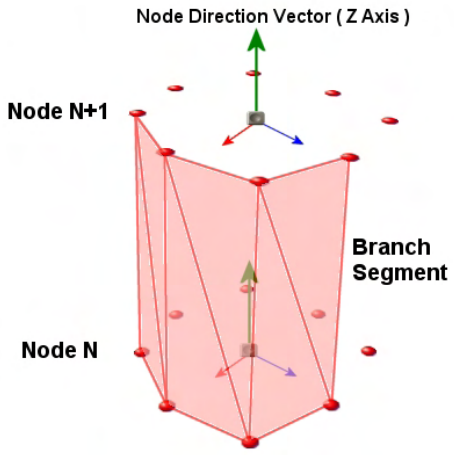


Figure 12.9

You might be wondering why each node would need its own direction vector if the tree is going to grow straight up most of the time anyway? Well, let us consider child branches that are spawned from a node of the trunk branch. They will almost certainly need to have their direction vectors deviated from the parent node's direction vectors, otherwise, all branches spawned would grow vertically straight up inside the trunk of the tree (we would not even see them).

One of the major aspects of the virtual tree generation process will involve deviation of a child's direction vector from that of its parent. For example, let us assume we start at the root node and assign it an initial 'straight up' direction vector of (0, 1, 0). Imagine that we continue to add child nodes with the same direction vector to the same branch, but then we hit a node where a random calculation says we need to add a new child branch. We know that this child branch must not share the same direction vector as the parent, so we will take the parent node's direction vector and rotate it (deviate it) by some amount. The maximum and minimum angles by which we deviate the child node's direction vector from its parent will be controlled by input parameters to the system. We will essentially use two angles to define a deviation cone. When a child branch is spawned and a new node is added to the parent node's child list, its direction vector will be generated by randomly deviating the parent direction vector within a specified range. As we have control over the deviation range via its input parameters, we can use fairly small ranges such that the child branches, while deviating from the parent direction, still follow the overall growth direction of the tree (appropriate for most trees). Alternatively, by specifying a larger random deviation range, we can have child branches growing off at wild angles and even coming back down against the initial growth direction of the root branch (useful for modeling some tree types).

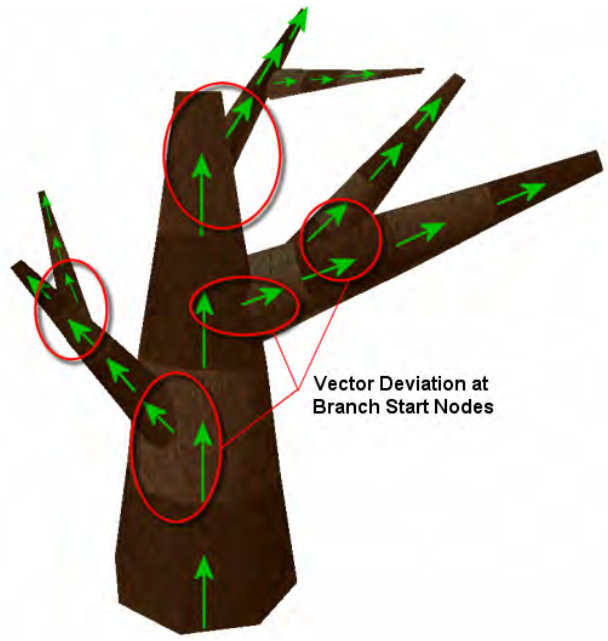


Figure 12.10 : Deviated Direction Vectors

Do not worry about how we deviate a vector within certain restraints as we will get to that in a moment. Figure 12.10 clearly shows why the nodes of each branch will need to have deviated direction vectors from that of its parent branch segments.

Looking at the tree in Figure 12.10, we see that it still is not quite there yet. While the branches shoot out at their own directions, the individual segments within a given branch are all still far too uniform. That is, all nodes within a given branch share the same direction vector. If we study the branch of a real tree, we know it is far from perfectly straight. Usually, it bends multiple times along its length. As we have already added the functionality to deviate a branch node's direction vector, we can widen the application of this technique to deviate every node in the tree from its parent's direction vector.

We will want to control the deviation such that, the deviation applied to a normal branch node's vector is not as strong as the one applied to a node that is the start of an entirely new branch. After all, if every single node in every single branch could deviate by a large amount, we might end up with branches that look more like springs. We can control this with two sets of deviation input parameters to the system. The first pair of input parameters describes the deviation cone that can occur between segments within the same branch. The second pair of parameters describes the deviation cone that can occur for the root node (the starting segment) of any branch. Typically, we will want the deviation cone used for segment deviation within the same branch to be quite small, as shown in Figure 12.11.

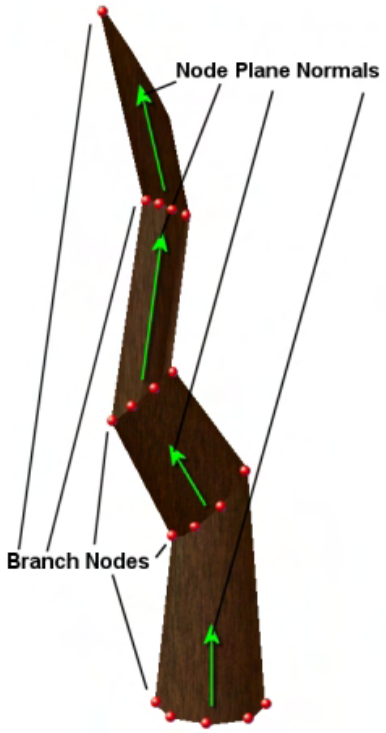


Figure 12.11

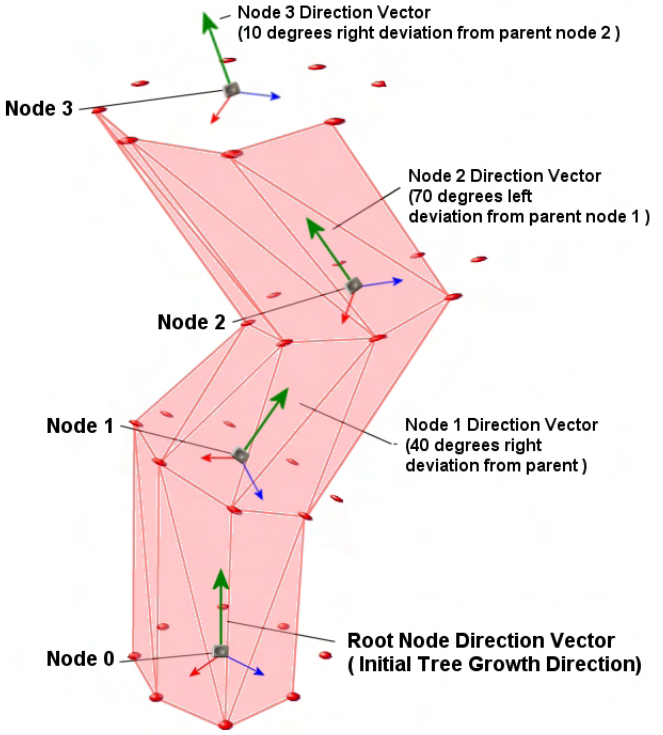


Figure 12.12

Note: Figure 12.11 and 12.12 show two different branches. They are not intended to depict the same vector deviations at each segment.

Figure 12.11 shows how a single branch mesh would be constructed if per-node direction vector deviation was used during the creation of the virtual tree hierarchy. In this diagram, you can see that the initial node had a direction vector (0,1,0) and thus a ring of vertices was inserted and aligned to the XZ plane. However, when a new node is generated to extend the branch, its direction vector is calculated by deviating the parent node by some random amount (within a range specified by input parameters). In this example, you can see that the second node deviated the parent node direction about 35 degrees to the left. Notice that when we come to insert the vertices for this node, the direction vector (along with the two tangent vectors), describes the plane on which the vertices must be positioned. Once again, we can imagine that the node position describes the initial point on the plane at which we add vertices, before then translating them out along the plane (using the tangent vectors) into their positions in the ellipse shape (see Figure 12.12). Looking at the third node (third row of vertices) in Figure 12.11, we can see that this node was deviated from its parent about 45 degrees to the right. This process is repeated for every node of every branch.

So we have learned that when we build our node hierarchy, we will start with the initial growth direction vector of the root node. This vector will be passed through a recursive procedure and deviated for the next node, which will pass its new direction vector onto its child node where it will be further deviated, and so on. Each iteration of the node hierarchy building process will involve, in simple terms, positioning a new child node, and calculating a direction vector for that node by deviating the parent vector.

When we combine the lesser per-segment deviation with the more extreme deviation performed to calculate the direction vector for each starting node in a new branch, we end up with a virtual tree representation that. when converted to mesh form, is a lot more pleasing (see Figure 12.13).

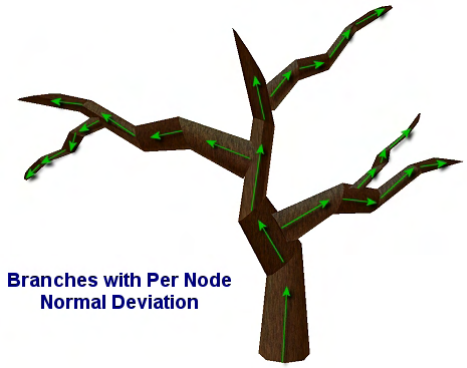


Figure 12.13

Figure 12.14 depicts how the node hierarchy for this tree might look after it has been created. For reasons of image clarity, we have slightly offset the nodes that start new branches from the position of their sibling nodes. In reality however, they would occupy the same position.

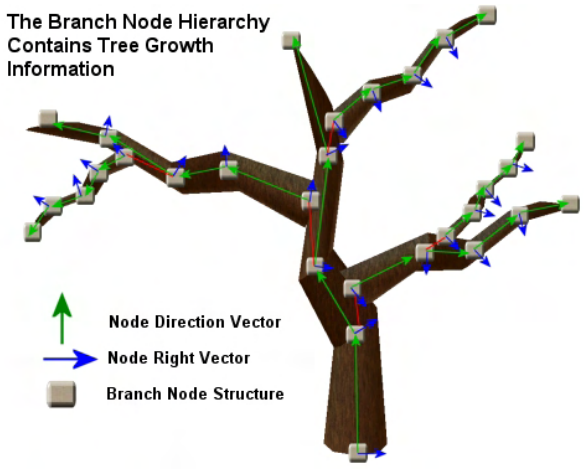


Figure 12.14

If you examine Figure 12.14 you will notice that each node contains a direction vector depicted by the green arrows, and a right vector that is perpendicular to the direction vector. We will also pass the right vector of the initial segment into the system in addition to the direction vector. Why do we need to pass this right vector through the tree? We need it in order to deviate the direction vector of each node during the recursive process. This will be explained in the next section.

12.2 Direction Vector Deviation

Before we examine the source code, we have one major process left to discuss on a theoretical level. Given a vector A, how do we generate a new random vector B that is contained within a specified cone set up around the original vector? It is easy in the two dimensional case since we merely have to rotate the new vector either left or right of the original vector by a random number of degrees. However, when working in three dimensions, simply rotating the vector left or right, or backwards or forwards will not allow us to generate a new vector that is anywhere within the specified cone.

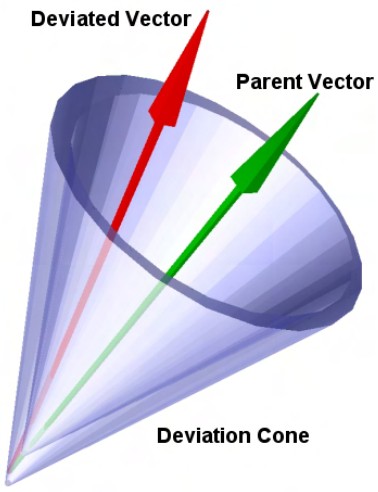


Figure 12.15 : We need to generate a random vector anywhere within the deviation cone.

Figure 12.15 demonstrates the problem we need to solve. The green arrow running down the center of the cone is assumed to be the parent vector (i.e., the vector we wish to deviate). The cone itself represents the range over which the vector can deviate. The goal is to generate a totally random vector anywhere within this deviation cone (the red arrow is one example of a possible solution). As can be seen, simply rotating the original vector clockwise or counter clockwise about a single axis will not provide us with a means to accomplish our objective.

You will see in a moment, when we discuss the structures that we are going to use in our code, how the size of this cone is specified using an angle. However, we will also have another variable that influences vector deviation called *deviation rotation*. Describing what this variable is will allow us to also understand the question at hand, "How do I generate a random vector with a cone?" The answer to this question will explain why we need to calculate a right vector at the root node and pass it through the recursive process. Each node will have its own right vector which will be used in the deviation process of its child nodes.

We will use a simple example to demonstrate the process of deviating the vector. While the same deviation function is used for the deviation of normal segment nodes and for nodes starting new child branches, in this example, we will show how a new branch is spawned from a node and how that node's direction vector is deviated to create the direction vector of the new branch start node. To give clarity to these examples, we will show the mesh data that will eventually be created from the virtual node. However, just remember that the node hierarchy that we generate will not contain any mesh data; it will be a hierarchy of data structures containing positional and directional information that will later be used to describe a tree shape to the mesh building process.

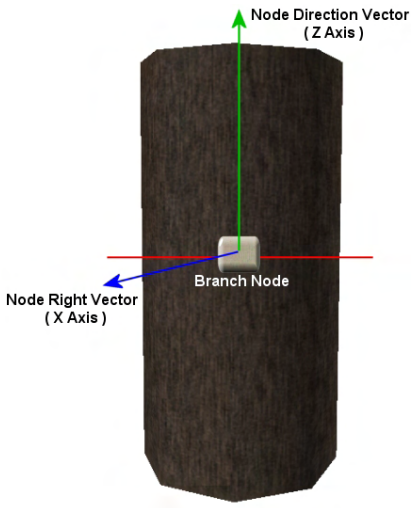


Figure 12.16

In Figure 12.16 we see a single node that is situated at some arbitrary position along a branch. We will assume this is a node in the root branch. Notice how the node stores a direction vector (the local Z axis of the node) and a right vector (the node's local X axis).

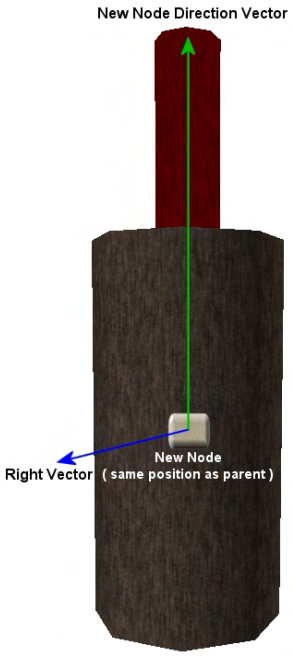


Figure 12.17

Next, let us assume that while processing this node we determine that we would like to create a new branch at this node. In Figure 12.17, the new node is added at the position of the parent node and currently has inherited its direction vector. In the diagram the red cylinder mesh sticking out of the top of the parent segment would not actually exist at this point, but will make our explanation easier. We can think of this for now as being the branch segment that will eventually be built from the branch start node we have just created.

In this diagram, the right vector of the parent node is assumed to be coming out of the page. You will notice that this provides an axis around which we can rotate the new branch node's direction vector. For example, if the cone deviation angle was 90 degrees, then we could generate a matrix that will rotate the new node's direction vector about the parent node's right vector by some angle between 0 and 90 degrees (to allow randomness). In fact, that still is not quite what we want since that would only rotate the vector one way (right for example). What we want to do instead is choose a random number between –cone angle/2 and +cone angle/2. For example, if our input parameters describe to the system that a new child branch node should be deviated by 90 degrees, we would generate a random number between –45 and +45 and use this value to build an axis rotation matrix around the parent node's right vector. This will allow for counter-clockwise and clockwise rotations within the cone.

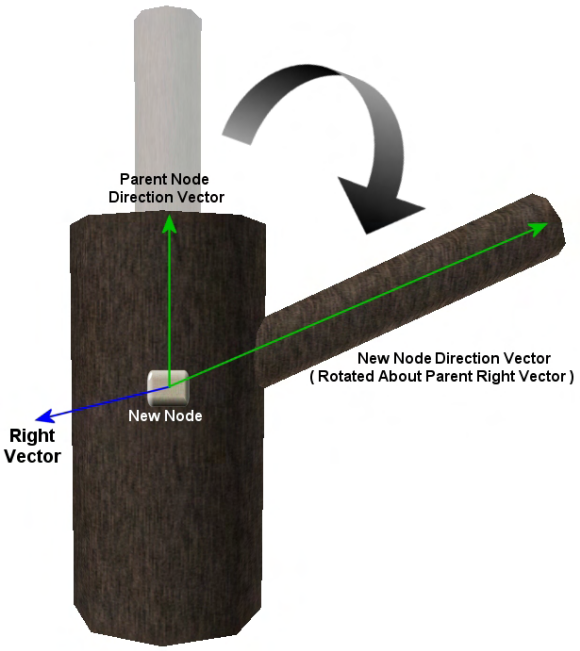


Figure 12.18

After rotating the child node's direction vector about the parent node's right vector (Figure 11.18), we will perform a second random rotation of that vector about the parent node's direction vector. This vector acts as a local up vector for the child node allowing us to rotate its newly rotated direction vector to any position around the circumference of the parent branch (see Figure 12.19). This means, our vector deviation routine will require an additional angle range that describes the permissible rotation that can be applied to the child node's direction vector when generating its random position on the branch. Usually, we will want to give branches an equal chance of shooting out from the parent at any angle, and as such, would set the up axis rotation range to 180 degrees. Then, when we need to rotate the child direction vector about the parent direction vector, we simply choose a random angle between 0-180 to rotate the new node into its final position.

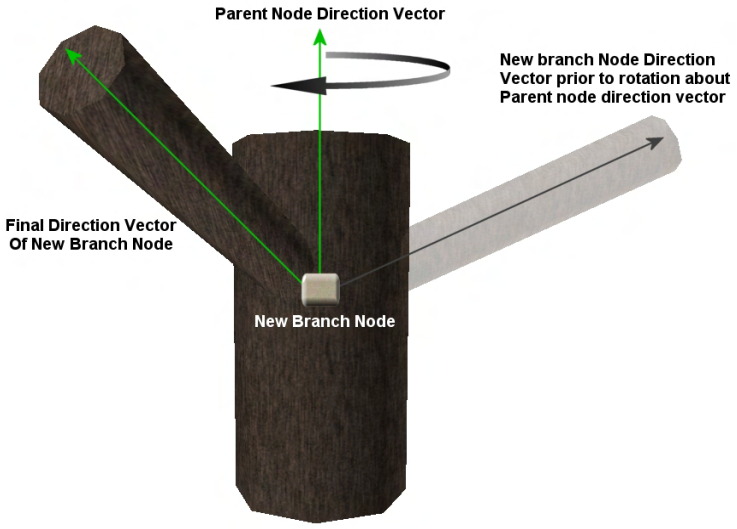


Figure 12.19 : Polar Rotation

Why do we only need to rotate 180 degrees in the second rotation step instead of 360? Remember that in the first step we will rotate the branch either counter-clockwise or clockwise depending on whether a positive or negative angle is used for rotation about the right vector of the parent node. Therefore, this first step essentially rotates the child node's vector into either the negative or positive half of a circle surrounding the parent node. Therefore, the first step chooses the semi-circle which the branch will grow from, and the second step allows us to access any random angle within that semi-circle. Of course, you can limit the rotation angle if you want branches limited to a smaller range of values on either side of the circle. Usually though, in the case of deviating the vector of a new branch start node, you will want to allow full 180 degree rotation on either side so that the branch can grow from anywhere. It should be noted that the same deviation procedure is used for deviating both the direction vectors of new branch start nodes and for the deviation between nodes within the same branch. However, as we will usually want segment to segment deviation within the same branch to more closely follow the overall growth direction of the branch, we will generally use much smaller deviation angles for both rotation steps in the process. With this two step deviation approach we essentially describe a virtual cone with the parent node's direction vector at the center. The new deviated vector will exist somewhere within this cone (Figure 12.20).

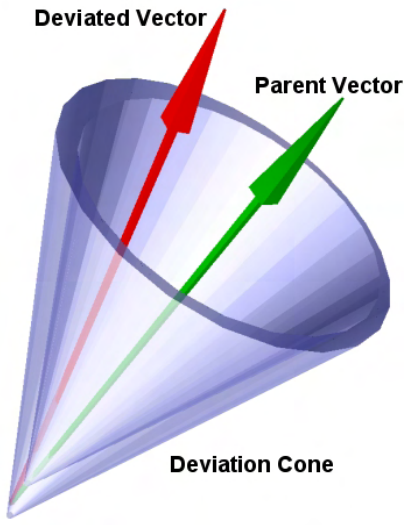


Figure 12.20

In the data structure that we feed into our CTreeActor::SetGrowthProperties function (prior to calling the CTreeActor::GenerateTree function), we will specify two sets of ranges. We will set an axis rotation range for both up and right vectors for node to node deviation within the same branch, and we will also set an axis rotation range for both of the vectors that the deviation process will use when calculating the direction vector of the node that is to become the first node in a new branch. By separating these into two separate deviation cases, we can easily control new branch and inter-branch deviation.

Looking at Figures 12.18 and 12.19 we can see that having the right vector stored at every node is important during the initial growth of the virtual tree as it is needed to deviate the directions of any child nodes. Therefore, while we have shown that the direction vector of each node must be deviated from the parent to create a new direction vector for the child node, once this has been accomplished, we must also calculate the right vector of the new node.

As it happens, updating the right vector in the deviated node is simple since we have already generated the matrix to rotate it into its new position. When we performed rotation on the child nodes direction vector, we would have rotated about the parent node's direction vector by some random amount. We will use 90 degrees in this example. So all we have to do is rotate the parent node's right vector around the parent node's direction vector and we have the new right vector for the child node. The basic node building process now looks like this:

We start the process by passing in a single growth direction vector for the tree (e.g., <0,1,0>). We will perform the cross product with this vector and the world axis that is least aligned to it to create the right vector. We now have the growth direction vector and the right vector for the first node in the tree. From this point on the recursive process starts and works as follows:

1) Will this node generate another child node in the same branch (another branch segment)?
    a) Generate a new segment node and attach it to the parent as a child
    b) Deviate the parent node's direction vector and right vector using (small random values)
    c) Store deviated vectors in the child node

2) Will this node generate a new branch?
    a) Generate a new branch node and attach it to the child list
    b) Deviate the new node's direction vector and right vector using (large random values)
    c) Store deviated vectors in new child node
    d) Repeat steps a-c for each new branch generated at this node
3) Repeat steps 1 and 2 for newly generated node(s)

When we set the deviation properties for our system, we will in fact have three range values we can specify for each of the two deviation types (segment deviation and new branch deviation). We will have a minimum cone angle, a maximum cone angle and the polar rotation angle. The minimum and maximum cone angles (which are a pair) are used to influence the first rotation step (the rotation around the parent's right vector). These values instruct the deviation function to generate a random deviation angle for the first rotation step that is no greater than the maximum cone angle, but also, no smaller than the minimum cone angle. This allows us to set a minimum level of deviation to ensure that we always get at least some deviation. When setting the minimum and maximum cone angle for the new branch deviation settings, the variables can greatly effect the shape of the tree. For example, by using a large minimum angle and only a slightly larger maximum angle, you can generate a tree that has a very uniform looking branch growth direction (like a conifer tree).

By introducing a minimum cone angle variable combined with the polar rotation step, we essentially specify a region inside the cone where vectors can be produced. It is perhaps more accurate to say that we define a region at the center of the cone where random vectors *cannot* be produced (Figure 12.21).

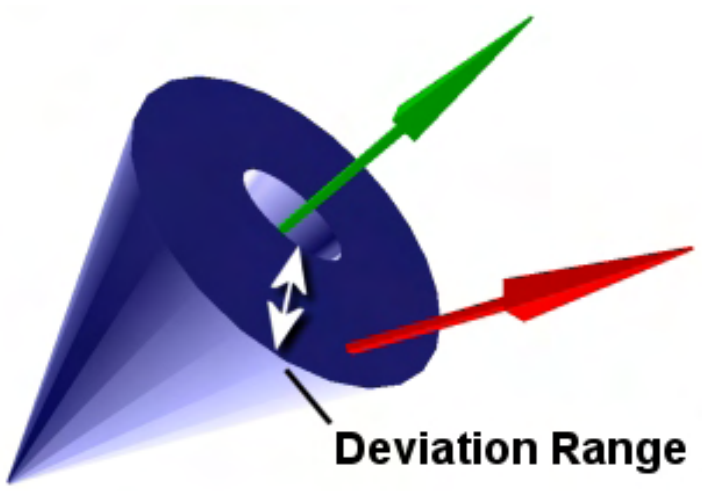


Figure 12.21

While this might sound like a complicated process, you will see shortly that the CTreeActor::DeviateNode method is only a few lines of code long.

12.3 Building the Virtual Tree

We now have all the theoretical knowledge at our disposal to discuss the code to the first part of the tree generation process. This will be the process that essentially grows the virtual tree one node at a time. Remember, at this point we are not dealing with mesh or bone data at all; we are simply building a hierarchy of data structures that contain direction vectors, right vectors, and dimensions.

Let us first see how the application might generate a tree using our new CTreeActor class.

```
// Create a new CTreeActor
CTreeActor * pTree = new CTreeActor;

// Fill out a TreeGrowthProperties Structure
TreeGrowthProperties tgp;

// Fill in the TreeGrowthProperties structure here
// NOT SHOWN HERE AS WILL DISUCSS THESE PROPERTIES IN A MOMENT

// Send out properties to the actor
pTree->SetGrowthProperties ( tgp );

// Build everything
D3DXVECTOR3 RootSegSize( 2.0 ,2.0 ,2.6 );
pTree->GenerateTree ( RootSegSize )
```

That is all there is to it. We create an instance of CTreeActor and then we fill in a TreeGrowthProperties structure. We will look at all the members in this structure in a moment. For now just know that it influences the way the nodes of the virtual tree are grown. We then send this structure to our CTreeActor so that it will have access to these properties during the tree generation process. Finally, we call the one function that makes it all happen: GenerateTree. This function first builds the virtual tree of information nodes (which we call *branch nodes*). Note that we pass in a vector of dimensions that describe the three radii of an ellipsoid that will contain the first branch segment (the root node). The X and Y components describe a box that will bound the ring of vertices generated for that node. If X and Y are equal, the branches will be perfectly circular. However, if we assign X and Y different values such as 2 and 5 for example, the thickness of the root segment of the root branch would be two tree space units in the X dimension and 5 tree space units in the Y dimension. As tree space is essentially the world space coordinate system, but with the tree at the origin, these dimensions directly describe the thickness of the root segment in world space (provided you do not transform the actor into world space using a scaling matrix). The Z component of the dimension vector describes the length of the root segment of the root branch. That is, when building the root branch (the trunk), this is the distance along the root node's direction vector that the second node will be positioned. This equates to the location of the second ring of vertices during the mesh building process. You will see when we examine the recursive tree building process, that the dimensions of each branch node are scaled down versions of the dimensions inherited from their parent. This ensures that branches get thinner and shorter as they near their end.

When the virtual tree has been completely built, we will then enter the second phase of tree construction. We will traverse the virtual tree hierarchy, creating vertices at each node and adding them to the meshes for the branch to which that node belongs. While building the meshes for each branch, we also assemble (using the node hierarchy) a D3DXFRAME hierarchy that will be used as the bone system for the tree.

Finally, when the actor's hierarchy and meshes have been built, we create the actor's animation controller and add optional animations to simulate the tree branches swaying in the wind. Once the GenerateTree function returns, the tree is complete and we can use it in our application just like a regular actor, or we can save it out to an X file for import it a world editor. Using the CTreeActor class within your application (should you wish not to save the generated tree out to an X file and load it in as a regular actor) is no different from using a normal actor. That is, you call its AdvanceTime method to update its animations each frame update and use its DrawSubset methods to render it. The virtual tree hierarchy is no longer needed at this time since it was only used to generate the frame hierarchy of bones and the mesh data for each branch.

12.3.1 The TreeGrowthProperties Structure

So that we get a better feel for the system, the best place to start is the TreeGrowthProperties structure since it defines the behavior of the CTreeActor during the virtual tree building process. This structure is pretty large as the system has many variables that you can tweak to provide you with a means to generate a vast number of different tree configurations. Not all of the members of this structure will make immediate sense until we see them being used. So a detailed discussion of some of them may be deferred until the actual mesh building process is covered.

```
typedef struct _TreeGrowthProperties
{
    USHORT      Max_Iteration_Count;
    USHORT      Initial_Branch_Count;
    USHORT      Min_Split_Iteration;
    USHORT      Max_Split_Iteration;
    float       Min_Split_Size;
    float       Max_Split_Size;

    float       Two_Split_Chance;
    float       Three_Split_Chance;
    float       Four_Split_Chance;
    float       Split_End_Chance;

    float       Segment_Deviation_Chance;
    float       Segment_Deviation_Min_Cone;
    float       Segment_Deviation_Max_Cone;
    float       Segment_Deviation_Rotate;

    float       Length_Falloff_Scale;

    float       Split_Deviation_Min_Cone;
    float       Split_Deviation_Max_Cone;
    float       Split_Deviation_Rotate;

    float       SegDev_Parent_Weight;
    float       SegDev_GrowthDir_Weight;
```

```
    USHORT      Branch_Resolution;
    USHORT      Bone_Resolution;

    float       Texture_Scale_U;
    float       Texture_Scale_V;

    D3DXVECTOR3 Growth_Dir;
} TreeGrowthProperties;
```

This structure may seem a little daunting at first, so we will examine the members one at a time. You will see that many of them relate to the concepts we have already discussed. They act as a means for setting ranges in which certain behaviors can and cannot happen during the growth process.

USHORT Max_Iteration_Count

The maximum iteration member allows us to control the overall depth of the virtual tree hierarchy we create. We start at the root node with an iteration count of 1. Every time a new segment/node is generated for a branch, the iteration count is increased and assigned to the child node. This recursive process continues such that, with every node within a given branch, the segment count is increased. As soon as (or if) the iteration count of the current node reaches the Max_Iteration_Count, no new segments/nodes will be generated along the branch and the branch will be capped and ended. One thing to watch out for is that when new child branches are spawned, the first node in that branch inherits the same segment count as the sibling node of the parent branch. Remember, if a node spawns three child nodes, there will be a sibling list of four nodes in the hierarchy -- three branch start nodes and the node that spawned the branches (the next segment of the current branch being processed). All nodes within the sibling list will contain the same iteration number (see Figure 11.22).

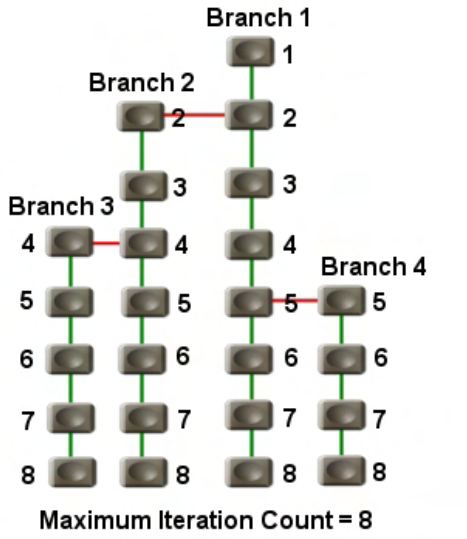


Figure 12.22

Figure 12.22 shows an example hierarchy for a tree that is constructed from four branches. The first branch is, of course, the trunk; but notice that when we add the second node of the trunk we also decide to start a new child branch. The second node of the trunk and the first node of the child branch are in a sibling list, both with an iteration count of 2. This iteration count is then continued down each branch. As you can see, the fourth branch starts with an iteration count of 5 and terminates at 8, being only four segments in length. In this example, we use a maximum iteration count of 8, so the virtual tree hierarchy will never be deeper to traverse than eight levels.

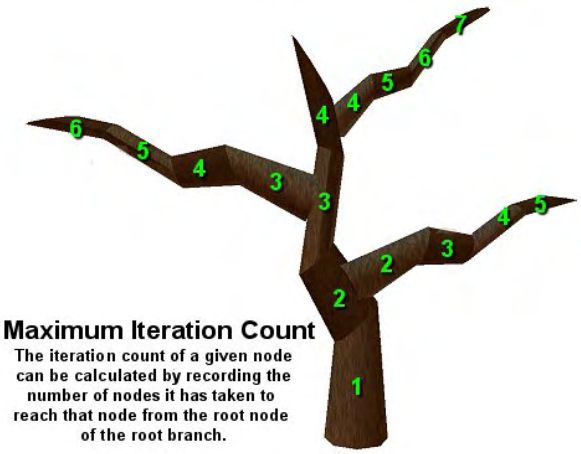


Figure 12.23

Figure 12.23 shows the tree in mesh form so that we can see how the iteration counts apply to the various segments. We are using a bit of artistic license here since we are placing the node iteration count values in the branch segments. We know that in reality however, the nodes that will contain those iterative values will be positioned at the branch segment boundaries (two of which create a branch segment). Nevertheless, this should provide an easy way to see how the iteration count can be used to set a maximum tree depth on the recursive procedure during the generation of the virtual tree.

It should be noted, that while Figure 12.22 clearly shows each branch terminating at the maximum iteration count threshold, this will not always be the case. There are many factors that are considered when deciding to add another segment/node to a branch or whether to terminate the branch at the current node. Such factors are the current branch thickness and of course a random element thrown in for good measure. Therefore, if we set the maximum iteration count to 20, we will not generate a tree where all branches end at the 20^{th} level in the hierarchy. Some branches may be terminated much sooner due to other factors. However, it is one of the many limiting factors that we will place on the procedure to control the depth of our hierarchy and the ultimate the size and complexity of our tree. There will never be a single path of branch segments from the root segment in the root branch to any branch's terminating segment that will cross segment boundaries more times than the value we set here.

USHORT Initial_Branch_Count

In all the example trees we have examined so far, we have assumed that the tree had a single initial branch (the trunk) from which all others were spawned. However, there is no reason why this has to be a strictly followed rule. While it is clear that your trees will require one initial branch (the trunk), you may wish to model trees that have multiple trunks (Figure 12.24). This parameter allows us to specify the number of initial branches we would like to tree to have.

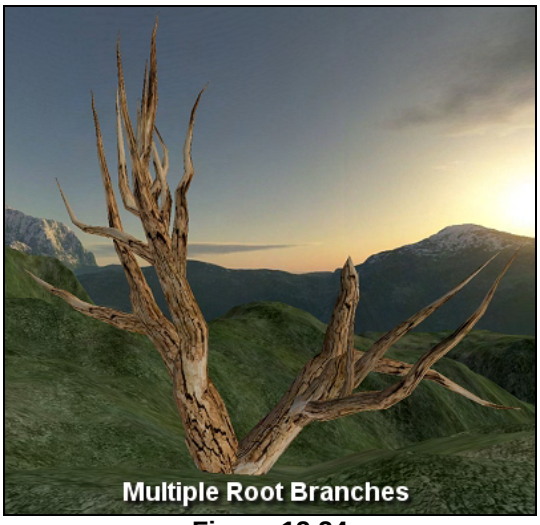


Figure 12.24

When multiple initial branches are enabled, all initial branches start at the same position (0,0,0) in tree space. Normally, when a single initial branch is being used, the direction vector we pass into the CTreeActor::GenerateTree function will be used 'as is' for the direction of the first node in that branch. When multiple initial branches are being used, each initial node in each trunk branch we create will be randomly deviated from the initial vector passed in. For example, we might imagine when looking at Figure 12.24, that when GenerateTree was called, an initial growth direction vector of <0,1,0> was specified. However, we cannot assign this same initial vector to the first node in each root branch or they would be created in exactly the same position and facing the same direction (thus, we would only see one branch). So instead, the passed vector is randomly deviated to create the actual initial direction vectors of each branch root node. In Figure 12.24 you can see that the node direction vector for both root branches has been rotated left or right to some degree from the initial direction vector passed in.

From the perspective of our virtual tree hierarchy, it simply means that there will no longer be only a single root node in the first level of the hierarchy (describing the start node of the only trunk). Instead, the first level of the hierarchy may consist of a sibling list of nodes, where each node in the list describes the start node of a trunk branch. By default, the Initial_Branch_Count member is set to 1 (in the constructor). By setting it to higher values, we are able to model foliage that grows in a more clustered manner (e.g., a rhubarb plant).

> **Note:** CTreeActor can be used to model foliage types beyond trees (bushes, shrubs, etc.) using the same methodology. If you did not want to include the overhead of all of the hierarchy and animation and skinning for small shrubs, you can still use CTreeActor to generate the shrub and then save it as an X file. When you load it back in, you can load it as a standard mesh (CTriMesh) rather than as an actor. We will talk more about this later.

USHORT Min_Split_Iteration
USHORT Max_Split_Iteration

These two members provide us with a means to control the range in which child branches are allowed to be spawned. For example, we will often *not* want a child branch to be spawned at the first segment of the trunk (during the first iteration of the recursive process).

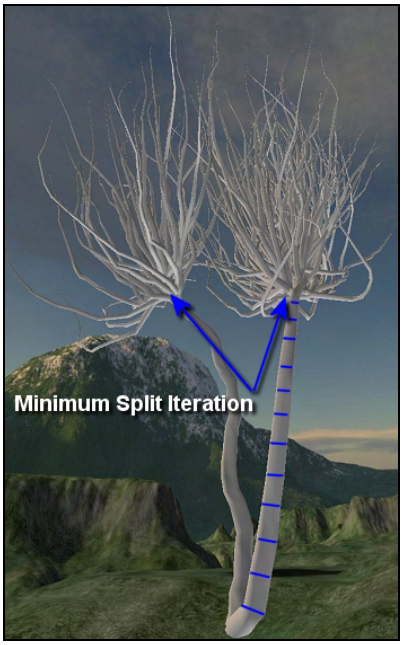


Figure 12.25

As discussed above (see Max_Iteration_Count), the iteration value is increased as each child node is added to the virtual tree hierarchy and it is directly related to the current branch segment that will be added to the mesh. We can think of the iteration count of a given node as indirectly describing the level in the hierarchy at which it resides. Figure 12.25 shows an example where we have set the minimum split iteration count quite near to the maximum iteration count of the tree such that splits into child branches only occur at the very top of the tree. Simply put, when a new node is added to the virtual tree hierarchy, we will decide whether or not we should split at this node and create multiple child branches. Splits will never happen if the iteration count of the current node being processed is smaller than Min_Split_Iteration or larger Max_Split_Iteration. Therefore, a node will only be considered for splitting if its iteration count (its depth in the hierarchy) is within the range described by these two members.

float Min_Split_Size
float Max_Split_Size

These two values allow us to further define a range of branch sizes with which child branches can be spawned from a node. When we call CTreeActor::GenerateTree, we will pass in a vector describing the size of the initial root node. As previously discussed, the X and Y components of this vector will describe the radii of the circle/ellipse that will be used to position the vertices at that node when we build the mesh. With each iteration, when we add new child node, we will subtract a small amount from this vector so that the vertices placed at each node define a smaller ellipse the further along the branch we move. This will produce a tapering effect. At the very end of a branch, the ellipse becomes a single point defining the tip of the branch. Essentially, by dividing the dimensions vector of the root branch node by the maximum number of iterations of the tree, we have a value that we can subtract from the dimensions of a parent node to generate the dimensions of the new child segment. By using this value, we can be sure that any branch will get progressively smaller as its length is traversed, with the final end node in a branch being the tip.

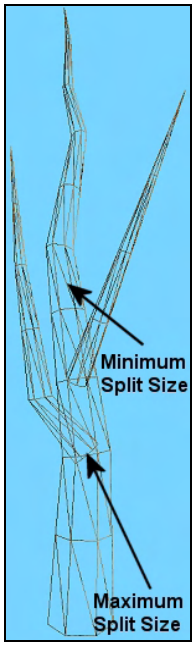


Figure 12.26

Near the end of a branch, its segments will typically be very thin, and as such, generating new child branches from these would result in extremely slim branches. They would be so thin that it would be a waste of time rendering them, especially when we consider that we will eventually add leaves to this tree that will completely obscure such microscopic detail. These two values allow us to define a thickness

range. If a node's dimensions are within this range, then it is possible for it to split into a child branch at this node. However, if the node thickness is smaller than the minimum split size or larger than the maximum split size, splitting the node will not be considered.

Figure 12.26 shows an example of a very simple tree where the Min_Split_Size and Max_Split_Size properties have been set to a very limited range in the center of the tree. As you can see, the Max_Split_Size variable allows us to control child branches not being spawned too close to the base of a branch, while the Min_Split_Size allows us to cease splitting the branch once its thickness has decreased past a sensible level.

These two members and the previous two members can be used together to provide a very flexible control mechanism for determining when and where child branches are spawned (and thus, control the shape and complexity of the tree).

float Two_Split_Chance
float Three_Split_Chance
float Four_Split_Chance

When building our virtual tree hierarchy, at every node we will need to decide whether or not we wish to introduce a new child branch. Of course, if the current iteration (tree depth) is such that it is outside the range described by the Max_Split_Iteration and Min_Split_Iteration members, then the node will not be considered for splitting. Furthermore, if the thickness of the node (its X and Y dimensions) is outside the range described in the Min_Split_Size and Max_Split_Size members, then once again, the node will not be considered for splitting. If however, the node is within the specified ranges for a possible split to occur, then we will use a random procedure to decide whether a split will occur at this node. In fact, we will perform three tests to decide whether the node should spawn two, three, or four branches (or just the one default).

These three properties each contain a percentage score that describe the probability of each node splitting into two, three, or four branches respectively. The tests are actually performed in order; we first perform a test to see if two branches should be spawned, then we perform a test to see if three branches should be spawned and finally we test to see if four branches should be spawned. The following pseudo code should give you the basic idea until we get the actual code.

```
// Generate Random Number Between 0 and 100
if ( RandomNumber < Two_Split_Chance)    NewNodeCount=2;
if ( RandomNumber < Three_Split_Chance)  NewNodeCount=3;
if ( RandomNumber < Four_Split_Chance)   NewNodeCount=4;
```

A random number will be generated between 0 and 100. Our three split chance members should be defined in that range too. Note above that all tests are always performed, so we always fall back to the highest number of splits should more than one of the tests succeed. For example, assume that we set our Four_Split_Chance member to 40 (40% chance) and our Two_Split_Chance to 80 (80% chance) and that at a given node the random number generated is 10. 10 is smaller than 80 so we have passed the two split chance and NewNodeCount is set to 2. However, when we perform the last test we also find that 10 is smaller than 40 so the Four_Split_Chance case wins out and we introduce four new branch nodes (three new branch segments plus the next branch segment of the current branch we currently processing). A different random number is generated for each test, so this is not as redundant as it

sounds. The reason we have ordered the test such that the highest number of splits takes precedence if multiple tests pass, is because typically we will set the probabilities of the four split case much lower than the two split case. Therefore, the two split case will often succeed when no others do. In the rare cases where the four split test does succeed, we want it to override the results of any previous tests.

By setting these values to different percentage values, we can influence how many branches are generated at each node. If all the tests fail, then no split will happen at this node and just the one node (the continuation of the branch) will be added.

float Split_End_Chance
This member is another percentage [0, 100] that describes the probability that each node has of ending the branch to which it belongs if it spawns one or more child branches. When it is determined that a branch node we are adding to the current branch will also spawn new child branches, we will generate a random number between 0 and 100 to determine whether the current branch we are adding should be terminated when the new branches are spawned. If the random number we generate is smaller than the Split_End_Chance probability, the current node will terminate the current branch, allowing the new child branches it spawned to continue. This value is not considered when a branch node is added that does not spawn new child branches. It really is just used to determine what the chances are of a branch terminating at the point where it forks into multiple child branches. We can think of it as the probability that the branch has been pruned at that node.

float Segment_Deviation_Chance
float Segment_Deviation_Min_Cone
float Segment_Deviation_Max_Cone
float Segment_Deviation_Rotate
These members control the range of random vector deviation that is applied to a child node's direction vector. As previously discussed, when a new node is added to continue a branch (not a branch start node), we will typically want to randomly deviate the direction of the new node so that all the segments in the branch do not end up sharing a perfectly uniform direction. The Segment_Deviation_Chance member should be set between 0 and 100 and will be used to determine the probability that a given node will deviate from its parent node's direction. If you set this to 100 for example, then a child node's direction vector will always be deviated from its parent node's direction vector making the branches of the tree look much more organic. Setting this to a value of 0 will generate a tree where all segments within the same branch share the same direction and no deviation will occur between segments within that branch.

Once it is determine that a child node's vector will be deviated, we will deviate it in two steps as discussed earlier in this lesson. First, we will rotate the child node's direction vector (initially inherited from the parent node) around the parent node's right vector. The amount we rotate this vector is described in degrees by the Segment_Deviation_Min_Cone and Segment_Deviation_Max_Cone properties. For example, if we set the minimum cone angle to 60 and the maximum cone angle to 110, then a random deviation angle will been chosen between 60 and 110 degrees. However, we do not always want the rotation to happen in the same direction around the parent node's right vector, otherwise our trees will have a tendency to all lean one way. So we convert the random angle into either a negative or positive number based on a random decision. This would allow us in this example to create a rotation between either –60 and –90 degrees or +60 to +90 degrees. Below we see the code that would

generate the rotation angle used in the first step. It generates a rotation angle between the minimum cone angle and the maximum cone angle in either the clockwise or counter-clockwise direction.

```
float fAzimuth = (float)rand() / (float)RAND_MAX;

fAzimuth = Segment_Deviation_Min_Cone +
          ((Segment_Deviation_Max_Cone – Segment_Deviation_Min_Cone) * fAzimuth);

if ( RandomNumber > 50% ) fAzimuth = -fAzimuth;
```

The first line generates a random number between 0.0 and 1.0. The second line essentially uses this value to scale the delta value between the minimum cone angle and maximum cone angle which is then added to the minimum cone angle. At this point we have a positive angle in the correct range, so the third line generates a random number between 0 and 100 and basically flips the sign if the random number is in the second half of its range. This ensures we have a 50/50 chance that each deviation will be either positive or negative in direction.

Let us plug in some values to see how that works. Let us imagine that we have set the minimum cone angle to 40 degrees and the maximum cone angle to 110 degrees. Let us also imagine that the initial random value (fAzimuth) is 0.5. This essentially means we wish to perform either a positive or negative rotation to a position halfway between 40 and 110 degrees (+/- 75 degrees).

```
float fAzimuth = 0.5; // Half way between min and max

fAzimuth = 40 +(( 110 – 40) * 0.5);
//       = 40 +(( 70) * 0.5 )
//       = 40 +( 35  )
//       = 75 degrees

if ( RandomNumber>50% ) fAzimuth = -fAzimuth;
```

That works perfectly. Once we have a value of 75 degrees in the above example, the second random number would be generated to decide whether or not to flip the sign and make this a +75 rotation or a –75 rotation.

As another example, if the random fAzimuth value we initially generated was 0.0, this means we wish to apply the minimum deviation. The minimum deviation we can perform is defined by the Segment_Deviation_Min_Cone angle. Using the same example values above, this was set to 40, which means the minimum we should deviate is 40 degrees. Let us plug it in and see if it works.

```
float fAzimuth = 0.0; // Apply minimum deviation

fAzimuth = 40 +(( 110 – 40) * 0.0);
//       = 40 +(( 70) * 0.0 )
//       = 40 +( 0  )
//       = 40 degrees

if ( RandomNumber>50% ) fAzimuth = -fAzimuth;
```

As you can see this would generate a rotation of either –40 degrees or + 40 degrees.

With the first deviation phase out of the way, our next task (having rotated the new node's direction vector left or right) is to rotate it in a circular fashion about the parent node's direction vector. This allows us to rotate the direction vector of the child so that is can protrude from the parent branch at any angle. Setting this value to 180 degrees will allow for a vector with total freedom of deviation in a 360 degree circle about the parent node.

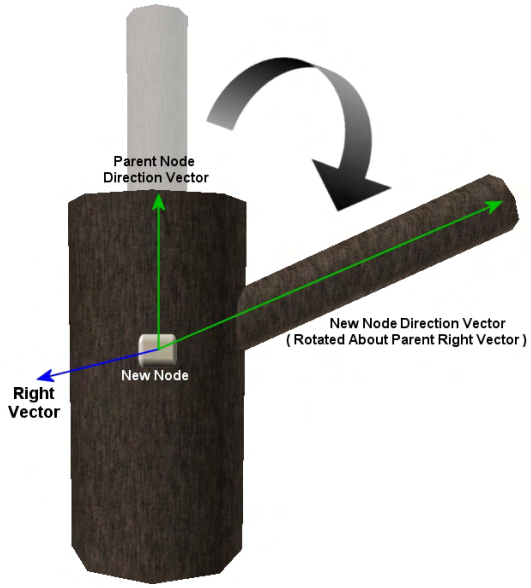


Figure 12.27
Phase 1: Random +/- rotation about parent node's right vector in the range defined by the minimum and maximum cone angles.

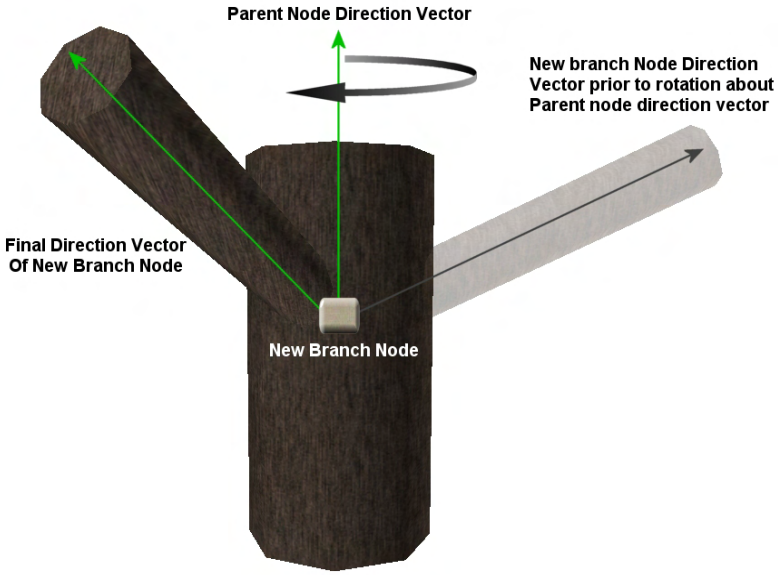


Figure 12.28
Phase 2: Random +/- rotation about the parent node's direction vector within the range: – (Polar Angle/2) to +(Polar Angle/2)

Once again, if this polar angle was set to 40 degrees, it would mean a random angle would be generated between 0 and 40. However, we do not always want to rotate about the parent node's direction vector in the same direction, so we will map it into the -/+ range. In this instance, we would want to generate a value between –20 and +20 instead of 0 to 40 using the code shown below.

```
fPolar  = (float)rand() / (float)RAND_MAX;
fPolar  = (Segment_Deviation_Rotate * fPolar) - (Segment_Deviation_Rotate / 2.0f);
```

To generate the random polar rotation angle we first generate a random number in the range of 0.0 to 1.0. We then use this value to generate an angle that is mapped into a -/+ range.

For example, let us imagine that we had set Segment_Deviation_Rotate to 30 degrees. Let us also assume that we generated an initial random number of 0.0. This should perform a negative rotation 15 degrees left as shown below.

```
fPolar  = 0.0;
fPolar  = (Segment_Deviation_Rotate * fPolar) - (Segment_Deviation_Rotate / 2.0f);
//      = ( 30 * 0.0 ) – ( 30 / 2 )
//      = 0.0 - 15
```

```
//      = -15 degrees
```

Likewise, an initial random number of 1.0 will generate a positive 15 degree rotation.

```
fPolar  = 1.0;
fPolar  = (Segment_Deviation_Rotate * fPolar) - (Segment_Deviation_Rotate / 2.0f);
//      = ( 30 * 1.0 ) – ( 30 / 2 )
//      =  30 - 15
//      =  15 degrees
```

Finally, an initial random number of 0.5 is halfway between the range and should therefore generate a rotation of 0.0 degrees (no polar rotation).

```
fPolar  = 0.5;
fPolar  = (Segment_Deviation_Rotate * fPolar) - (Segment_Deviation_Rotate / 2.0f);
//      = ( 30 * 0.5 ) – ( 30 / 2 )
//      =  15 - 15
//      =  0 degrees
```

And that is all there is to node deviation. We then use these two rotation values to build the rotation matrices with which to rotate the direction and right vectors of the child node.

float Length_Falloff_Scale

This member is one of those members that will make more sense when we look at the code. It controls how the length of each segment gets smaller as the segments near the end of the branch (the tapering effect). As discussed above, as we add each new node to our hierarchy, each node inherits the dimensions of its parent node which then has some small value subtracted from it. In the case of the X and Y dimensions of a node, these describe the radii of an ellipse on the plane described by the node, and thus the size of the ring of vertices that will be generated there. We also discussed how the value we subtract from each node is the dimensions of the root node divided by the maximum iteration count. This allows us to scale the radii of each node such that the ellipse of vertices inserted at each node will get progressively smaller towards the end of the branch (and thus the overall tree). We also do the same for the Z dimension of each node, which essentially describes the length of the cylinder segment formed by that node and its child node (the length of a branch segment). This means branch segments will not only get thinner as the tree approaches its branch tips, but also shorter.

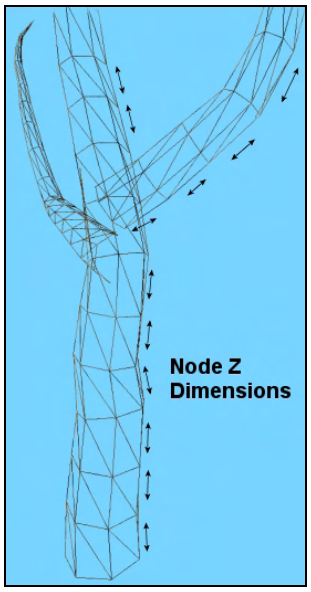


Figure 12.29

However, if you look at Figure 12.29, you can see that while we usually wish the thickness of the branch to get smaller towards its end quite quickly, usually we will not wish to scale the length of each segment at quite the same rate. In Figure 12.29 you can see that the length of each segment diminishes only slightly from segment to segment while the thickness of the each segment (the X and Y dimensions) falls off quite quickly.

The Length_Falloff_Scale allows us to control how the length of each segment (the Z dimension) gets decreased from node to node with respect to how the X and Y dimensions are scaled. For example, if we

set this value to 1.0, then the length of each segment in a branch will get smaller by the same ratio as the reduction in the thickness from segment to segment. A value of 0.5 would mean that the reduction in length from segment to segment would be half the reduction in branch thickness from segment to segment. Lower values in this member make the tree more spindly looking (long and thin) while higher values will result in tree with shorter, stumpy looking branches.

float Split_Deviation_Min_Cone
float Split_Deviation_Max_Cone
float Split_Deviation_Rotate
These values should look familiar to you. They define the cone and polar rotation used to deviate the vector of a node that is the first node in a new branch. The exact same deviation technique is used as has been previously described, and as such, these parameters are used in exactly the same way as the Segment_Deviation_Min_Cone, Segment_Deviation_Max_Cone and the Segment_Deviation_Rotate members discussed earlier. These values are used when deviating the vectors for start nodes of a new branch. This allows us to provide a much larger deviation range for new child branches which will usually sprout off from the parent at quite arbitrary angles. Contrast this with segment to segment deviation within the same branch where we usually keep vector deviation more conservative.

float SegDev_Parent_Weight
This member is used to set a weight (usually between 0.0 and 1.0) which describes how much the deviated vector of a child node should be influenced by the direction vector of its parent. Essentially, once we have deviated the vector of a node, we will add to that vector the direction vector of the parent scaled by this weight, before normalizing that vector.

```
DeviatedVector        = This is the vector that has just been randomly deviated
pNewNode->Direction   = DeviatedVector+( pParentNode->Direction*SegDev_Parent_Weight );
D3DXVec3Normalize    ( &pNewNode->Direction, &pNewNode->Direction );
```

If this value is set to zero then segment to segment vector deviation will be completely random and sporadic (within the specified ranges). By assigning this weight a value, we allow each new segment added to a branch to be randomly deviated while still following the overall direction of the branch.

It is very important to realize that the parent nodes direction vector and this weight are only used to influence the deviated vector of a child node during segment to segment deviation within the same branch. Whenever a new branch is generated, the direction vector of the first node in that branch (the root node of the branch) is not influenced by its parent node's direction vector at all. This makes sense as we often want new branches to shoot off at random angles from the parent branch. However, once the new random vector for a branch start node has been generated, all child segments/nodes of that branch will have their vectors influenced by their parent direction vectors, the first of which is the direction vector of the branch's root node. Therefore, new branch nodes are the links where parent influence is temporarily discarded (for the generation of that node only).

If you were to set this weight value to 1.0, the final vector for a new node would be the average of the deviated vector (generated in the steps discussed previously) and the parent node's direction vector.

D3DXVECTOR3 Growth_Dir
float SegDev_GrowthDir_Weight

These two members further allow us to control the overall growth direction of the tree. When we call CTreeActor::GenerateTree we pass in a vector describing the direction of the first node in that tree. From this point on, the direction vectors are passed from parent to child and deviate at each step. Therefore, even if you passed in an initial direction vector of <0, 1, 0>, this does not mean that your tree would grow in that direction at all. This only tells us that the first segment will point in that direction. As discussed in the previous parameter, we can certainly factor in the parent vector of any node into the generation of its child node's vector, however this does not always do what we want.

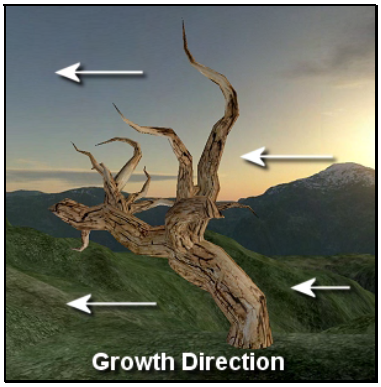


Figure 12.30

For example, if we used an initial direction vector of <0,1,0> for the root node and also set the SegDev_Parent_Weight to some non-zero value, we know that the direction vector would influence all the other segments in the root branch to some degree. This means, the root branch would generally grow in an upwards direction as described by our initial direction vector. However, what we must consider is that as soon as a new branch is spawned, the influence of the root parent branch direction vector is lost. That is, if a new branch is generated, the first node in that branch will have a totally random deviation applied to it (not influenced by its parent). From that point on, all the child nodes of that branch will be influenced by that direction vector and not the vector that we initially passed in. This is not a design flaw, since we absolutely want branches to have directions independent of their parent branches. But it would be nice if we had another level of control; a direction vector that could globally influence the direction of all nodes in the hierarchy. That is what the Growth_Dir and the SegDev_GrowthDir_Weight properties are for. These values are very useful for shaping the overall growth direction of a tree.

If you take a look at Figure 12.30 you can see a tree that has been generated with a growth direction vector of <-1,0,0> to make the tree segments generally grow in the direction of the negative X axis of tree space. In this example, a SegDev_GrowthDir_Weight of 0.2 was used. As you can see, even though we passed in an initial direction vector of <0,1,0> for the root node, as the segments are generated, the growth direction vector is still influencing the direction of every branch node. Once again, the only time that the growth direction does not influence the vector generated for a node is when the node is a root node of a branch. For example, you can see in Figure 12.30 how initially, new branches deviate in a totally random direction. As we begin to add nodes to those branches however, those child nodes are influenced by the growth direction vector and as such, the branches begin to curl round in the direction of the growth vector specified.

Let us say for example that you wanted to model a tree growing out of the side of cliff face. The growth direction for the tree could be set to <0,1,0> but the initial direction vector of the root node of the entire tree set to <1,0,0>. The tree would start growth horizontally out of the cliff face and then gradually start to grow upwards. How quickly this change in growth direction takes place depends on how you set the SegDev_GrowthDir_Weight member and the SegDev_Parent_Weight members. These members directly control, at each node, how strongly the parent node influences the node direction, and how strongly the overall growth direction of the tree does.

USHORT Branch_Resolution

This member is used to control how many vertices will be used in our node rings (and thus, our whole tree). As mentioned previously, after we have generated our virtual node tree, we will traverse that tree and insert a ring of vertices at each node. How many vertices we insert into this ring directly controls smoothness of the branches. Figure 12.31 shows the circle of vertices inserted for two nodes with a branch resolution setting of 8. As you can see, the Branch_Resolution member defines how coarse and angular the cylinder of each branch segment will be.

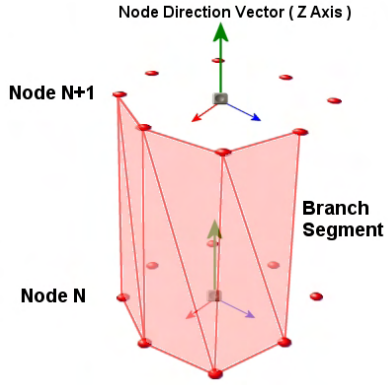


Figure 12.31

The default value is 8, which means each branch segment cylinder will be using 16 vertices -- 8 vertices at each node that form its bottom and top boundaries. Note however that vertices are shared between segments. For example, the ring of vertices inserted at node 2, form the top of cylinder 1 (Node1 ->Node2) and the bottom of cylinder 2 (Node2->Node3).

USHORT Bone_Resolution

Once we have completed the first phase of tree creation, we will have a Virtual Tree described by a hierarchy of branch node structures. This tree representation is abstracted from mesh or bone data at this point. The second phase of tree creation will involve generating the meshes for the branches themselves and building the actor's internal D3DXFRAME hierarchy which will describe the bones of the tree used to render and animate the actor.

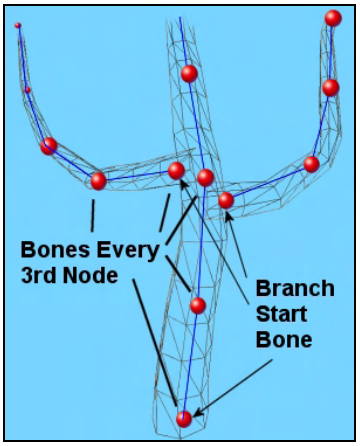


Figure 12.32

When building the meshes for the tree, we have decided that every node in the virtual tree should describe a plane that will contain a ring of vertices that both end and begin each branch segment. Therefore, each node in our virtual tree represents the actual location where vertices will be placed. We could take the same approach when building the actor's frame hierarchy, although having bones placed at every node is probably overkill for what will simply be an animation that gently moves the tree back and forth in the wind. The more bones we create for the tree, the more traversal and transformation will have to be done when rendering the tree and animating the hierarchy. Therefore, this member allows us to define the bone resolution of the tree.

The bone resolution is simply a value N that describes to the tree creation process that bones should be inserted every N nodes. The default bone resolution is 3 which means, starting from the root node, bones

will be inserted every third node that is encountered along a branch. If you look at Figure 12.32 you can see how the bones might look if placed at every third node. We have found during testing that this provides more than enough joints to move the tree in a convincing manner. In fact, the tree will still move nicely even with fewer bones than this, so you could set this member based on the end user's system to get better performance if you intended to use CTreeActor in your actual game code. As you can see by looking at Figure 12.32, the actor's frame hierarchy will contain a lot fewer frames than the node hierarchy of the virtual tree that we create. Therefore, we can think of the actor's frame hierarchy as being a discrete version of the virtual node hierarchy. This is similar to how curved surfaces are eventually turned into a discrete polygonal representation based on a continuous curve. The more polygons you allow to model the curve, the more closely the final mesh will resemble the mathematical model of that curve. While our virtual tree hierarchy is not nearly as abstract as a curved surface, it does allow us to de-couple the virtual tree generation process from the bone and mesh resolutions we ultimately end up using the build the final representation.

float Texture_Scale_U
float Texture_Scale_V

The final two members in our structure will be used to set the scale of the texture used to map the branch meshes. We learned in Module I of this series how we can assign texture coordinates values outside the [0.0, 1.0] range to allow us to repeat a texture image multiple times over the face of an object. For example, we know that if we have a texture mapped to a quad that has its texture coordinates in the 0.0-4.0 range along both its U and V axes, the texture will be mapped to the surface of the polygon sixteen times. It will be mapped four times along the U axis and four times along the V axis.

In Figure 12.33 we can see a quad that has a texture mapped to it once because its UV range is in the [0.0, 1.0] range both horizontally and vertically. In Figure 12.34, the same quad has texture coordinates in the range of [0.0, 4.0] which tiles four rows and four columns of the texture image.

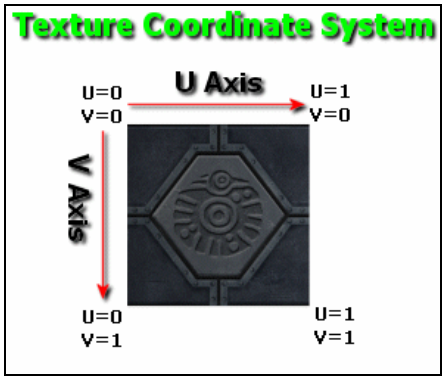


Figure 12.33 (UV range 0-1)
Texture mapped once to a quad.

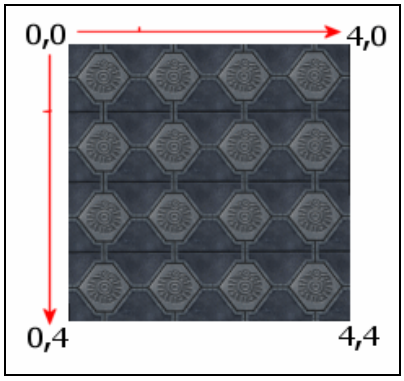


Figure 12.34 (UV range 0-4)
Texture mapped sixteen times to quad

The Texture_ScaleU and Texture_Scale_V properties of our tree allow us to apply the same scaling of the texture that we use for the branches of our tree. That is to say, the higher we set these values, the more times the texture will be tiled over the entire range of the tree. In order to understand how these properties work, we need to discuss how we will in fact generate the texture coordinates for the vertices of the tree.

Calculation of the U texture coordinate for a vertex in our branches is delightfully easy because it is a function of the branch resolution (the number of vertices we will use to form the ring at each node in our virtual tree). If the branch resolution is 8 for example, then we simply wish to assign each vertex in the ring one of 8 positions across the U axis of the texture. If we forget about the texture scaling members for the time being, we could see that the calculation of the U texture coordinate for a vertex in the ring is simply its position within that ring divided by the total number of vertices in the ring (branch resolution)

generating a value in the [0.0. 1.0] range. For example, if we have a branch resolution of 8, then we simply have to do the following calculation for each vertex in a ring:

```
for (int i=0; i< BranchResolution; i++)
{
     Vertex[i].U = i / BranchResolution-1; // zero based vertices
}
```

Let us have a look at the values this generates for each vertex in the ring.

Vertex[0].u = 0 / 7 = 0.0
Vertex[1].u = 1 / 7 = 0.142
Vertex[2].u = 2 / 7 = 0.285
Vertex[3].u = 3 / 7 = 0.428
Vertex[4].u = 4 / 7 = 0.571
Vertex[5].u = 5 / 7 = 0.714
Vertex[6].u = 6 / 7 = 0.857
Vertex[7].u = 7 / 7 = 1.0

As you can see, we have essentially unrolled the ring of vertices at that node and laid them flat across the texture. If you imagine those vertices then being pulled back into the shape of a cylinder, we have in fact wrapped the texture around the cylinder's width exactly once.

Calculating the V coordinate for each vertex is a slightly different matter but certainly no more difficult. We decided that the V coordinates should be mapped over the entire range of the tree. Essentially, vertices at the bottom of the tree will have V coordinates of 0.0 while vertices at the ends of branches will have V coordinates approaching 1.0.

We decided to do this because it allows us control over changing the color of the tree as a function of height. We could for example, have a bark texture that is very dark brown at the bottom but slowly faded to a light brown near the top. When mapped to the tree, the segment at the bottom of the tree would receive the dark brown texture colors, but as we move up the tree, the branch segments would receive the lighter portions of the texture. This might be to simulate that the branch tips are in direct sunlight and their color has been bleached. You could also generate a texture such that at the top of the texture, the bark texture has little green shoots or leaves, which once again, when mapped to the tree, the greener areas would only show up towards the branch ends.

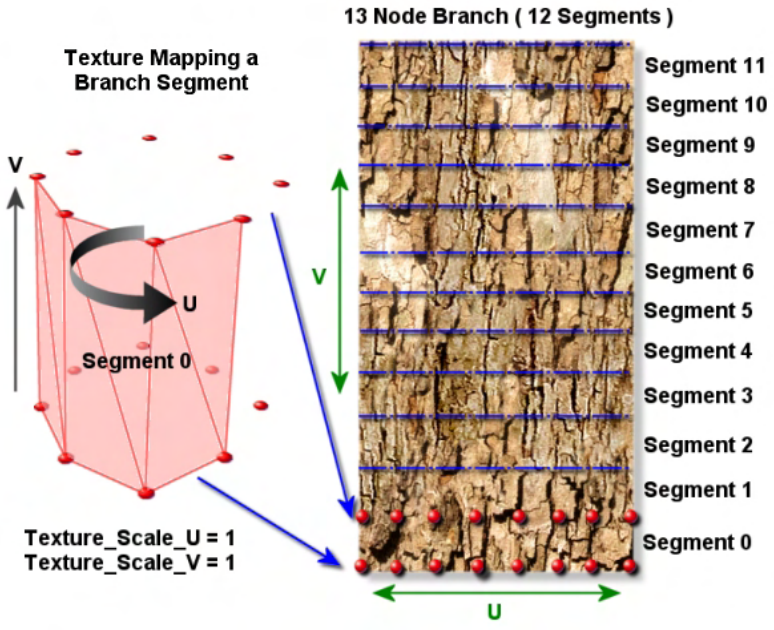


Figure 12.35
U Scale =1 : V Scale =1 : Max Iteration Count=13

In order to achieve the mapping of vertices such that the texture is mapped vertically over all segments of the tree, the generation of the V coordinate for each ring of vertices at a node, should be a function of that node's iteration count (its level in the virtual hierarchy) divided by the total hierarchy depth (maximum iteration count of the tree). Figure 12.35 shows this relationship clearly, which currently assumes a U and V texture scale of 1.0 (no scaling).

In Figure 12.35 we are shown the mapping of a texture to the first two rings of vertices (the first two nodes in the tree) which form the first segment in the root branch of the tree. For simplicity, we will also assume this tree consists of just a root branch which is constructed of 13 nodes (12 segments). If you look at the base of the bark texture, you can see how the two rings of vertices are mapped horizontally across the face of the texture using the U calculation texture we described previously. However, we can see that each node's vertices should be assigned different V coordinates which increase as the iteration of the node approaches the maximum iteration count of the tree. Using this technique, the V coordinates of the two rings of vertices forming the root segment are assigned low V values. It is also clear by looking at the segment boundaries marked on the texture that as we move up and calculate the V coordinates for the nodes deeper in the hierarchy (higher in the tree) the higher portion of the texture is mapped to the vertices of these nodes.

> **Note**: We have inverted the direction of the texture coordinate system V axis in the diagram so that it is more intuitive for this explanation. As we know, the V axis from runs top to bottom and not from bottom to top as shown here, so lower branch segments in the tree would actually be assigned the higher portions of the texture shown in this diagram, and the opposite is true.

By studying what we wish to achieve in Figure 12.35, the solution is easy. The V coordinate of any vertex is simply the iteration value of the node divided by the maximum iteration count of the tree (a property that we discussed earlier). The following snippet of pseudo code demonstrates the calculation of texture coordinates for a ring of vertices at an arbitrary node. In this example, the vertex structure is assumed to have a pointer to the branch node from which it was created. We will not implement it in this way, but this is to clearly establish the relationship between the node in the virtual tree and the vertices that are being created for it. In a moment when we look at the BranchNode structure, you will see how each node contains the iteration value that was generated during the virtual tree creation process and describes the level of that node in the virtual tree hierarchy.

```
for (i=0; i< BranchResolution; i++)
{
   Vertex[i].U = i / BranchResolution-1;
   Vertex[i].v = Vertex.pNode->Iteration / Max_Iteration_Count;
}
```

So we now know how to generate the texture coordinates for the vertices of all the branches in our tree. But what are the Texture_Scale_U and Texture_Scale_V members of the TreeGrowthProperties structure used for?

As you might imagine, they are simply used to multiply the result of the two texture coordinate calculations shown above so that we can tile the texture over the tree with a desired regularity. This upgrades our calculation code to the following final UV calculation.

```
for (i=0; i< BranchResolution; i++)
{
   Vertex[i].U = (i / BranchResolution-1) * Texture_Scale_U;
   Vertex[i].v = (Vertex.pNode->Iteration/Max_Iteration_Count) * Texture_Scale_V;
}
```

If both scaling factors are set to 1.0 then no texture scaling is done; a ring of vertices is simply mapped once horizontally across the face of the texture, and all the vertices comprising the tree are mapped once vertically along the face of the texture. By multiplying the texture coordinates by some scale value, we push them outside the 0.0 to 1.0 range, which we know will cause the texture to tile (unless texture tiling has been disabled in the API).

For example, we know that if we set the Texture_Scale_U property to 3.0, then each segment of a branch would have the texture's width wrapped around it not once, but three times. Obviously, for this to look good you must use a seamless tile-able texture. By performing scaling in this manner, the tree texture will look much higher resolution. If we were to set the Texture_Scale_V property to 2.0, then the texture would be mapped twice over the entire height of the tree. In other words, if we had a tree consisting of a single branch of 16 nodes, the entire vertical range of the texture would be mapped in its entirety to the first 8 nodes of the tree and repeated again for the second 8 nodes.

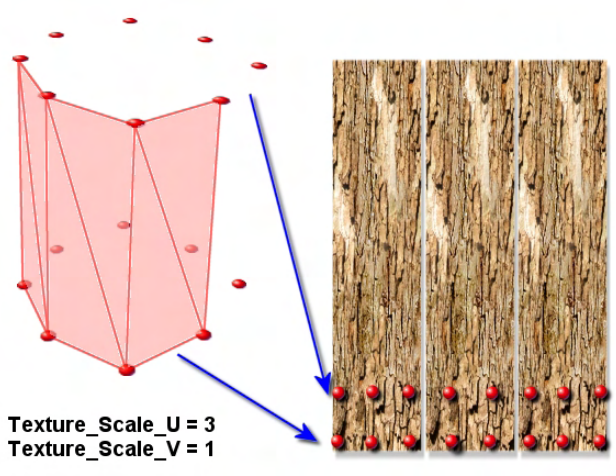


Figure 12.36

Figure 12.36 shows a simple example of tiling a bark texture with a Texture_Scale_U value of 3 and a Texture_Scale_V value of 1 (no scaling vertically). Once again, we are showing the vertex mapping for the first two rings of vertices (first segment) of the root branch. Notice how the 8 vertices now span three copies of the bark texture.

Figure 12.37 shows screenshots of the same section of the root branch of a tree mapped with this texture using different U and V texture coordinate scale values. Notice that the texture being used here is not a great texture for tiling but actually aids us in teaching this subject as it is clear that the texture is repeating across the width and height of the tree.

We can certainly see while examining the three example mappings in figure 12.37 how the U_Scale value is wrapping the texture multiple times around the section of the branch we are looking at. Of course, we are only looking at one side of the tree so we cannot see all the repeats. However, if you look at the center image, we can see a pattern starting to repeat which is much more obvious in the rightmost image. As mentioned, this texture is not ideal for tiling. But what should be clear is the extra detail that tiling seems to give the surface of our branches. The left most image, with no scaling being performed, looks quite blurry and out of scale by comparison. Of course, setting the correct texture scaling values in the TreeGrowthProperties structure will often only bear fruit by experimentation. Some textures look great when applied with no scale and some textures look awful when repeatedly tiled. Detail mapping can also be used to significantly improve the look and feel of the tree (see Module I).

U_Scale = 1 V_Scale = 1 | U_Scale = 5 V_Scale = 5 | U_Scale = 10 V_Scale = 10

Figure 12.37

We have now discussed the members of the TreeGrowthProperties structure. While that may have seemed daunting, remember that most of the last section was spent discussing how the tree will use these values in its construction. So by discussing this structure in such detail, we have also learned a great deal about how the tree will be built. This will make our job a lot easier when we examine the code. Remember, these values are set via the CTreeActor::SetGrowthProperties function, which simply assigns them to internal member variables which are accessible during tree creation. It is important that we set any tree properties before we issue the call to CTreeActor::GenerateTree since it is this function that builds the tree using the growth property information. Calling CTreeActor::SetGrowthProperties will have no effect if called after the CTreeActor::GenerateTree function.

12.3.2 The BranchNode Structure

When our application calls CTreeActor::GenerateTree method, the first task of this function is to build the virtual tree hierarchy. The nodes of this hierarchy will be arranged like D3DXFRAME structures in a frame hierarchy in that each node will contain a pointer to a linked list of children and another pointer to a linked list of sibling nodes that share the same parent. We are certainly used to this hierarchical arrangement by now. What is very important to grasp is that the hierarchy will not contain any mesh or bone data. Each node in the hierarchy is simply a packet of information, describing the position, orientation and scale of the tree at certain points. Later, this hierarchy of nodes will be used to build the actual meshes of the tree. As previously discussed, each node in this hierarchy will represent a branch segment boundary where a ring of vertices will be placed. Each node in our hierarchy will be represented by a BranchNode structure (defined in CTreeActor.h) whose members we will discuss momentarily.

It might strange that we would not build our virtual tree hierarchy out of D3DXFRAME derived structures. After all, this is the structure we are used to using and it has the child and sibling pointers that we need. It would at first seem an ideal choice to use for the nodes of this hierarchy. It is true that we will need to store much more information than a vanilla D3DXFRAME structure, but we could derive a class from this structure and add the extra members. So why do we not do this? There are several reasons we have decided to use a completely new structure for the building of our virtual tree.

The **first** reason is clarity. After our virtual tree has been constructed, we will traverse this hierarchy and build the actor's skeleton. We already know the actor's skeleton is just a D3DXFRAME hierarchy where each frame structure represents a bone. Therefore, we will be using one hierarchy (the virtual tree) to build another hierarchy (the actor's skeleton). If we implemented both of these as D3DXFRAME hierarchies, we might introduce confusion as to which hierarchy a given frame belongs to. By using two completely different hierarchy types, we eliminate the potential for bugs that could arise from accidentally connecting a D3DXFRAME structure to the wrong hierarchy.

The **second** reason is consistency. Although we can use the D3DXFRAME structure to store whatever information we choose, typically the matrix of this structure describes the position and orientation of that frame as a parent relative transformation. In order to convert that into an absolute transformation we must combine the matrices of all frames that precede it in the path down the hierarchy. We will be building our virtual tree hierarchy one node at a time using a random process and we certainly do not want to be adding nodes in parent relative space at this point. It makes much more sense to assign all the nodes a position and direction in a space shared by all nodes (tree space) so we are not constantly having to traverse the tree and perform transformations from one node's space to another. In short, we wish to store absolute positions and orientations at each node (not parent relative ones). Now it is true that we could simply use the matrix of each D3DXFRAME structure in the virtual tree hierarchy to store absolute transformations in this instance; but that is not consistent with what an application would usually expect the matrix of this structure to contain and may lead to incorrect assumptions about the matrices stored in this hierarchy if accessing frames directly.

The **third** and final reason is ease of use. It makes a little extra work for ourselves if we represent the position and direction of each node in matrix form during the building process. We have seen how we often need to work with the direction and right vectors in isolation when calculating the direction of a node and the deviation of its child nodes. Therefore, rather than having to extract the position and direction vectors from the matrix, modify them and store them back in the matrix, we will just store them in separate vector variables in the branch node structure.

The BranchNode structure also has a constructor and destructor which take care of initializing its member variables to zero and deleting its child and sibling lists, respectively. Below, we see the BranchNode structure and then discuss its member variables. Each node in our virtual tree will be a structure of this type.

```
typedef struct _BranchNode
{
    D3DXVECTOR3     Position;
    D3DXVECTOR3     Direction;
    D3DXVECTOR3     Right;
    D3DXVECTOR3     Dimensions;
```

```
    BranchNodeType  Type;
    _BranchNode    *Parent;
    _BranchNode    *Child;
    _BranchNode    *Sibling;
    USHORT          Iteration;     // The iteration at which this was generated
    USHORT          BranchSegment;
    USHORT          VertexStart;
    ULONG           UID;
    bool            BoneNode;
    LPD3DXFRAME     pBone;

    // Auto Hierarchy Destructor
    ~_BranchNode() { if ( Child ) delete Child; if ( Sibling ) delete Sibling; }

    // Auto clearing constructor
    _BranchNode()  { ZeroMemory( this, sizeof(_BranchNode) ); }

} BranchNode;
```

Let us discuss those member variables.

D3DXVECTOR3 Position
This member is where the tree space position of the node will be stored. We can think of tree space as a coordinate system where the root node/nodes of the trunk branch exist at the origin. The position, direction vector and right vectors of a node define a plane on which the ring of vertices that the node represents will be placed.

D3DXVECTOR3 Direction
This vector will contain the direction of the node. We can think of this vector as pointing in the direction of the next segment (child) in the branch. The direction vector can also be thought of as a normal to a plane on which the ring of vertices that this node represents will be placed.

D3DXVECTOR3 Right
This is where we will store the right vector of the node. This is a vector that is tangent to the plane on which the vertices will be placed and describes the direction in which the node's local X axis is pointing in tree space. That is to say, the Direction vector and the Right vector of a node represent the local coordinate system of the node (the Z and X axis respectively). By performing the cross product on these two vectors we can generate the third axis (the Y axis) of the local coordinate system for that node. This is used during mesh creation to position the ring of vertices on the node's plane.

D3DXVECTOR3 Dimensions
This vector stores the dimensions of the node. The X and Y components of the vector represent the two radii of an ellipse that has the node's position at its center. This is used to place the ring of vertices in a circle surrounding the node's position. The Z component of the vector describes the distance along the node's dimension vector to its child node (if it exists). We can think of the Z component of node N as defining the world space length of the branch segment (cylinder) formed by nodes N and N+1.

BranchNodeType Type
Each node in the tree will be one of three types of node which we must distinguish between when deviating vectors and building the mesh data. A node can either be a Branch Begin node, which means it

is the first node in a new child branch. It may alternatively be a Brand End node which means it is the terminating node in a branch and the node at which we will create only a single vertex (instead of a ring of vertices) forming the tip of the branch. Most nodes in the tree will be nodes of the third type: Branch Segment. These are nodes at some position in a branch between the branch begin and end nodes. It will become obvious when we cover the code why we must distinguish between the three node types in many places during the tree building process.

This member will be assigned one of three members of the BranchNodeType enumeration, which is defined in the CTreeActor namespace as:

```
enum BranchNodeType { BRANCH_BEGIN = 1, BRANCH_SEGMENT = 2, BRANCH_END = 3 };
```

_BranchNode *Parent
_BranchNode *Child
_BranchNode *Sibling
These three pointers are used to connect the branch node to the hierarchy. The Child and Sibling pointers mirror the functionality of the D3DXFRAME structure members of the same name. The Child pointer points to the first branch node in a linked list of child nodes in the next level of the hierarchy. If this node had a child node which had three siblings, this pointer would point to a linked list of four branch nodes. The parent node's Child pointer points to the child node at the head of the linked list, and each of the child nodes are connected by their sibling pointers. Likewise, the Sibling pointer will point to branch nodes that share the same parent node. The Parent pointer is not available in the D3DXFRAME structure and simply points to the node's parent branch node. This allows us to traverse up and down the levels of the hierarchy with ease from any given node.

As shown in Figure 12.22, the child pointer will point to a linked list of sibling nodes in the next level down in the hierarchy. This child list contains the next node in the current branch and any Branch Begin nodes of branches that start at that child node. The Sibling pointer will point to a list of nodes that exist at the same level and share the same parent. For example, let us imagine that node N in a branch has a child pointer to node N+1 which represents the next node in the branch. Also imagine, that at node N+1 two child branches are started. Node N would have a child pointer that would point to a linked list of three nodes: Node N+1 in the current branch and the two Branch Begin nodes for the new branches spawned at node N+1. These nodes would be connected via their sibling pointers.

USHORT Iteration
As discussed previously, as we step through a branch adding child branch nodes, the iteration count will be incremented and passed to the child nodes. The iteration basically describes the level in the hierarchy that the node exists at. For example, Node N would have an iteration count of N and any child nodes would have an iteration value of N+1, and so on down the tree. The Iteration of a node is also used in calculating its V texture coordinate as discussed in the previous section.

USHORT BranchSegment
This member will contain the Branch Begin relative index of the node. In other words, it stores the zero based node index from the start of the branch to which it belongs. Whenever a new branch begins, a new Branch Begin node is added with a BranchSegment value of zero. The next node of this branch will have a value of 1 assigned to its BranchSegment value, etc. We can think of this in many ways as being the branch local equivalent of the node's Iteration member. The node's iteration describes the number of

nodes that would have to be traversed from the root node of the root branch to reach the current node in the tree. The BranchSegment value describes the number of nodes that would have to be traversed from the first node (Branch_Begin node) in the current branch to reach the current node. This value will be useful when generating the mesh data for a given branch. Remember, each branch will be a separate skinned mesh.

USHORT VertexStart
This variable will be used when adding the vertices to at each branch node to aid in the building of branch segment indices. It will contain the vertex index where the current node's ring of vertices begins in the branch mesh. Remember that each branch will be a separate mesh, so this value will be the index where the ring of vertices starts in the branch mesh's vertex list. We will need to know this when adding branch segments to our mesh.

A branch segment is a cylinder of faces formed from the vertices at the parent node and the vertices at this node. To build these indices we will need an easy way of knowing the position at which a node's ring of vertices is placed in the branch mesh's vertex buffer. If a node had this value set to 50, and the branch resolution property of the tree was set to 8, we would know when generating the indices that this node's vertices will be positioned at 50 through 57 in the branch mesh's vertex buffer. This is a value we will generate when adding the vertices to each branch mesh in the second phase of creation.

ULONG UID
Every branch node in the virtual tree will contain a unique ID that identifies the node. This will start off at a value of zero for the first node of the hierarchy and will be incremented for each new node generated. The UID of the N^{th} node created will simply be N-1 because we start at 0 for the root node. So if the UID of a branch node had a value of 45, this would mean it was the 46^{th} branch node created during the virtual tree creation process. Why would we need to know this?

We will need this information when we build the skeleton for the actor in the second phase of tree creation. Certain nodes in the virtual tree hierarchy will become bones in the actor's frame hierarchy and as we know, frames that we wish to animate must be assigned names. We also need frames to have names in order to set up the skinning information. When a branch node is determined to be a candidate for a bone, we will need to give that frame a name. We also need to know that the name we assign the frame is unique with respect to any other frames in the hierarchy. We will assign a name to each frame using the format Branch_N where N is the UID of the branch node from which the bone is being created. Therefore, if we determine during the creation of the actor's skeleton that a branch node with a UID of 231 is to be used to create a bone for the actor's frame hierarchy, the name given to that frame will be Branch_231, which is guaranteed to be unique from any other frames we add.

bool BoneNode
When we discussed the TreeGrowthProperties structure, we examined a member called Bone_Resolution. By default it was set to 3. We discussed how this defines a ratio describing the number of bones we should create compared to the number of branch nodes in our virtual tree. The default value 3 means that we will create a bone for every third node in the branch node hierarchy. When building the actor's frame hierarchy (after the virtual tree of branch nodes has been constructed), we will traverse the branch node hierarchy starting at the root node. The Branch Begin node of every branch will always have a bone created for it. From that point on, we will create a bone using every third

node in the branch. Branch End nodes will never have a bone constructed from it, as it makes little sense to stick a joint at the tip. This second phase is where the frame hierarchy will be constructed.

The bones that we create from branch nodes between the Branch_Begin and Branch_End nodes form the skeleton for that mesh. Remember, each branch is a separate mesh and therefore will have its own bones connected into a larger hierarchy of bones for the entire actor. That is, the actor will have a skeleton that will represent the bones of the entire tree, but any given branch mesh will only use a localized set of those bones.

After the branch node hierarchy has been constructed in the initial phase, we can traverse that hierarchy and easily calculate which branch nodes should become bones and add them to the actor's frame hierarchy. This second phase is where the actor's frame hierarchy gets constructed from the virtual tree. Whenever we decide to make a bone from a branch node, we will set its BoneNode boolean member to true. This will be useful later on when building the skins for the various branches because we will need the original information stored in the branch node to calculate the bone offset matrix for a given bone in the hierarchy. It allows us to easily traverse a branch and find the branch nodes that have been used to construct bones.

As the BranchSegment member of a branch node contains the zero based index of that node within a given branch (a mesh), we can easily determine if a branch node will be a bone during the building of the actor's frame hierarchy. Simply put, every time the modulus of the branch node's BranchSegment member and the BoneResolution member equals zero, we have skipped the correct number of bones and it is time to mark the next new node as a bone node and create the accompanying frame in the hierarchy. Example code to determine if a newly created node (pNode) is a bone node is shown below.

```
if ( pNode->Type == BRANCH_BEGIN ||
   ((pNode->BranchSegment % m_Properties.Bone_Resolution)==0 && pNode->Type!=BRANCH_END))
{
        pNode->BoneNode = true;
        …
        …
}
```

As you can see, if this is a Branch_Begin node, then it is a node that starts a new branch and will become the first (root) bone of that branch mesh. In such a case we always make it a bone node. Otherwise, the only time we make a node a bone node is when the modulus of the local branch index of the node and the bone resolution wraps back around to zero (i.e., we have skipped the correct number of nodes). Notice as well that this is only true if the node in question is not a Branch_End node.

LPD3DXFRAME pBone

This member is also used during the creation of the actor's frame hierarchy in the second phase, and as such, is related and set using the same conditions as described for the previous member. As discussed, after building the virtual tree, we will enter the second phase where we construct the frame hierarchy (the skeleton) of the actor. We will traverse the branch node hierarchy and determine that certain nodes are bone nodes. For each one that we find, we will allocate a new D3DXFRAME, calculate its relative matrix based on the information stored in the branch node, and assign the frame (bone) a name before attaching it to the actor's hierarchy. We will then assign the new frame pointer to the source branch node's pBone member and set the branch node's BoneNode boolean to true. This establishes a

connection between the frame we have just added to the hierarchy and the branch node it was originally created from. This connectivity information will be used later when we manually build the ID3DXSkinInfo object for the branch mesh.

We have now discussed all there is to discuss about the structures used by CTreeActor and the properties that the tree generation process will use. While this has been an awful lot to take in without any code to look at, it is important that you understand the process overall before viewing the highly recursive code.

12.4 Source Code Walkthrough - CTreeActor

It is now time to look at our CTreeActor class, which is derived from CActor. Below you can see how the class is defined in CTreeActor.h. We will discuss the various new member variables of the actor in a moment. We will not immediately discuss the various methods of the class now as we will cover each one when we encounter it in the general flow of explaining the tree creation process.

```
class CTreeActor : public CActor
{
public:

    enum BranchNodeType { BRANCH_BEGIN = 1,BRANCH_SEGMENT = 2,BRANCH_END = 3 };

    // Constructors & Destructors for This Class.
             CTreeActor( );
    virtual ~CTreeActor( );


    // Public Functions for This Class
    void                SetGrowthProperties (const TreeGrowthProperties & Prop );
    TreeGrowthProperties GetGrowthProperties ( ) const;
    void                SetBranchMaterial ( LPCTSTR strTexture,
                                            D3DMATERIAL9 * pMaterial = NULL );

    HRESULT   GenerateTree( ULONG Options,
                            LPDIRECT3DDEVICE9 pD3DDevice,
                            const D3DXVECTOR3 & vecDimensions,
                            const D3DXVECTOR3 & vecInitialDir
                                              = D3DXVECTOR3( 0.0f, 1.0f, 0.0f ),
                            ULONG BranchSeed = 0 );


    HRESULT                 GenerateAnimation   ( D3DXVECTOR3 vecWindDir,
                                                  float fWindStrength,
                                                  bool bApplyCustomSets = true );

    virtual void            Release             ( );

private:

    // Private Functions for This Class
```

```
    HRESULT     GenerateBranches    ( const D3DXVECTOR3 & vecDimensions,
                                      const D3DXVECTOR3 & vecInitialDir
                                          = D3DXVECTOR3( 0.0f, 1.0f, 0.0f ),
                                      ULONG Seed = 0 );
    void        BuildBranchNodes    ( BranchNode * pNode,
                                      ULONG & BranchUID, ULONG Iteration = 0 );

    bool        ChanceResult        ( float fValue ) const;
    void        DeviateNode         ( BranchNode * pNode,
                                      float fAzimuthThetaMin,
                                      float fAzimuthThetaMax,
                                      float fPolarTheta = 360.0f ) const;

    HRESULT     BuildFrameHierarchy ( ID3DXAllocateHierarchy * pAllocate );
    HRESULT     BuildNode           ( BranchNode * pNode,
                                      D3DXFRAME * pParent,
                                      CTriMesh * pMesh,
                                      const D3DXMATRIX & mtxCombined,
                                      ID3DXAllocateHierarchy * pAllocate );

    HRESULT     BuildSkinInfo       ( BranchNode * pNode,
                                      CTriMesh * pMeshData
                                      LPD3DXSKININFO * ppSkinInfo );

    HRESULT     BuildNodeAnimation  ( BranchNode * pNode,
                                      const D3DXVECTOR3 & vecWindAxis,
                                      float fWindStrength,
                                      LPD3DXKEYFRAMEDANIMATIONSET pAnimSet );
    HRESULT     AddBranchSegment    ( BranchNode * pNode, CTriMesh * pMesh );


   // Private Variables for This Class
    ULONG                   m_nBranchSeed;
    TreeGrowthProperties    m_Properties;
    BranchNode             *m_pHeadNode;

    D3DMATERIAL9            m_Material;
    LPTSTR                  m_strTexture;
};
```

While the list of new member variables we have added is extremely small (only five), do bear in mind at all times that this class is derived from CActor and thus has access to all of its member variables and functionality as well. That is why you cannot see (for example) a member variable that points to the root frame of the tree's frame hierarchy. As you know, the frame hierarchy member variables (and the method to traverse them) are all inherited from the base class (CActor). Let us now discuss these new member variables one at a time.

ULONG m_nBranchSeed

As you certainly aware by now, the assembling of our tree will call for a lot of randomly generated numbers. For example, we will generate random angles for direction vector deviations between nodes in a branch and we will also use random numbers to decide whether or not a branch node currently being processed should spawn a new child branch. You will see when we cover the code in a moment that random number generation is used literally throughout the entire tree generation process.

We generate random numbers using the rand function (part of the standard C runtime libraries). Of course, a computer cannot generate a truly random number since this is essentially like asking the computer to make a real choice. Only sentient beings that are self aware can make random choices. So, the rand function (and its sibling function, srand) just generates and returns numbers using a simple calculation that returns a string of numbers that seem random to a human being. All the rand function is actually doing is performing a calculation where the srand function is used to set an initial value (a seed) that is used in that calculation. Each time the rand function is called, the initial seed value we set is updated to new value which is then used to influence the value returned in the next rand call. For example, lets us imagine that the C runtime library stores some global value called 'next' which is initially set to 1. That is, the default seed for the rand function is 1.

```
unsigned long int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int) (next/65536) % 32768;
}
```

As you can see, the rand function does not generate anything randomly. The first time it is called the 'next' variable will be set to the seed value (1 by default). The rand function simply multiplies this value by a very large number. Notice however that the 'next' value is updated with the result so that each time the rand function is called a different value will be returned. It then simply returns the number divided by 65536 and has a modulus performed with 32768 to snap the value into the correct range of an unsigned integer.

We use the srand function to seed the C random number generator (i.e., change the initial value of the 'next' variable in the above code). What is vitally important to note is that this list of pseudo-random numbers generated will *never* change. So if we use the same seed at the start of our application, the random numbers generated will be exactly the same every time the application is run. After all, every time the application is run, the same starting point in that list of random numbers is used. Thus ten calls to the rand function will generate the same ten random numbers every time the application is run. In terms of our tree, this means regardless of how many times we use the rand function in our generation process, our GenerateTree function will create the same tree each time the application is run. This may or may not be desirable.

Often it is not desirable. If a game uses random numbers to generate random events, we do not want these events to happen on cue every single time at the exact same place every time the application is run. Therefore, usually the value used to seed the random number generator is taken from a value that is unique every time the application is run (e.g., the timestamp of the current system time or the number of microseconds that have passed since the computer was switched on). We take such a value and use the srand function to seed the random generation process. Below we can see that the call to srand simply sets the initial value of the 'next' value used in the rand calls. If we use a different seed every time the application runs (system time for example), the random numbers generated will be different every time also.

```
/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

That fact that using the same seed will generate the same string of pseudo-random numbers can actually be beneficial in our tree creation process. When the application calls the CTreeActor::GenerateTree method we can optionally pass in a seed value that will be used to seed the random number generator prior to tree creation. We will use the timeGetTime function call to seed the random number generator if the application passes no seed. As this value will change each time a tree is generated, we will get different trees every time the function is called. However, you also have the option to pass in any seed value that you like so that the same tree is generated. This allows you to continually generate trees using different seed values until you find a tree that you really like. You can then remember those seeds and use them to generate the trees for your game, knowing exactly what the trees that will be generated will look like.

The m_nBranchSeed member of the class will contain the seed that was used to generate the tree. This will be either the seed value passed in by the application or the value returned from timeGetTime. If you see a tree that was randomly generated that you really like, you can examine the contents of this member and use this seed in the future to create that same tree.

TreeGrowthProperties m_Properties
This member contains the creation properties used to generate the tree. We set the members of this structure using the CTreeActor::SetGrowthProperties and can also retrieve it using the CTreeActor::GetGrowthProperties methods. We have already discussed all the members of this structure in detail..

BranchNode *m_pHeadNode
This member stores a pointer to the root branch node of the virtual tree hierarchy that was generated during tree creation. Remember, this is not the root frame of the actor's hierarchy (stored in a base class member variable). The hierarchy of branch nodes is used to grow a virtual tree representation and then used to build the frame hierarchy and meshes of the actor.

D3DMATERIAL9 m_Material
LPTSTR m_strTexture
Our branches will need to have a texture and material applied to them. We set these values using the CTreeActor::SetBranchMaterial function. Notice that we do not pass in an actual texture, only the filename. Essentially, this is the same information our actor is passed by D3DX when we load a material from an X file. This is certainly by design because from the CActor's perspective, there will be no difference. What do we mean by this?

After we have generated our virtual tree we will step through that hierarchy and generate the frame hierarchy similarly to how D3DX does it when we call D3DXLoadMeshHierarchyFromX. Every time we wish to allocate a new frame for example, we will call the actor's CAllocateHierarchy::CreateFrame method, just as D3DX does during the loading process. Our CAllocateHierarchy object is already written, so we use it again in exactly the same way. Similarly, every time D3DX encounters a mesh

during the loading process, it will call our CAllocateHierarchy::CreateMeshContainer method which is responsible for loading textures and materials (via registered callbacks) and for generating the actual skinned mesh. We will follow this exact same technique so that all the code we have developed in that function can be re-used without modification to generate our tree meshes.

When we traverse the virtual tree hierarchy generating mesh data, we will do two things. We will generate a regular mesh and we will also populate an ID3DXSkinInfo object with bone information. Once we have done that, we will simply call the CAllocateHierarchy::CreateMeshContainer method passing the regular mesh we have generated for the branch being processed and its skin info. We will also pass it the material and the texture filename that has been set for the tree. As we know, the CreateMeshContainer method will generate a skinned mesh from the regular mesh passed in and will also take care of calling any callback functions to handle the loading and processing of textures and materials. From the perspective of CreateMeshContainer, nothing is different from the case where the mesh data is being loaded by D3DX. We still pass it a regular mesh, texture filename and material structures, the same format in which the D3DXLoadMeshHierarchyFromX provides the mesh information.

It is actually quite a complex relationship when you examine it. In our application, our CTreeActor is set to non-managed mode prior to tree generation (the application registers an attribute callback function). As we know, this callback function is responsible for loading textures from filenames and storing those textures and their accompanying materials in the scene database. We also set the texture filename and the material the tree should use immediately afterwards. Next we make the call to the GenerateTree method to generate the actual meshes. So, for each branch mesh we create, the texture filename is passed to the CreateMeshContainer method which then sends that same texture filename back to the application defined callback which loads the actual texture used by the actor. So, the application supplied the actor with a texture filename which is later passed back to its callback function where it will load that texture.

Constructor - CTreeActor

The constructor of CTreeActor simply initializes the tree's growth properties to some default values that will be overridden as soon as the application calls the SetGrowthProperties method. It also sets a default material in case the application does not call the SetBranchMaterial method to specifically set one. The constructor is shown below. Please note that the values assigned to each of the tree growth properties may differ in the actual source code. At the time of this writing, the development team was still experimenting with their preferred defaults.

```
CTreeActor::CTreeActor()
{
    // Call base constructor
    CActor::CActor();

    // Reset all required values
    m_nBranchSeed = 0;
    m_pHeadNode   = NULL;
    m_strTexture  = NULL;
```

```
    // Setup some useful initial growth properties
    m_Properties.Max_Iteration_Count        = 21;
    m_Properties.Initial_Branch_Count       = 1;
    m_Properties.Min_Split_Iteration        = 2;
    m_Properties.Max_Split_Iteration        = 20;
    m_Properties.Min_Split_Size             = 0.8f;
    m_Properties.Max_Split_Size             = 20.0f;

    // 25% Chance of splitting in two
    m_Properties.Two_Split_Chance           = 15.0f;

    // 25% Chance of splitting in three
    m_Properties.Three_Split_Chance         = 5.0f;

    // 1% Chance of splitting into four
    m_Properties.Four_Split_Chance          = 1.0f;

    // 5% Chance that a branch will end when a split occurs.
    m_Properties.Split_End_Chance           = 5.0f;

    m_Properties.Segment_Deviation_Chance   = 60.0f;    // 60% chance of deviation
    m_Properties.Segment_Deviation_Min_Cone = 0.0f;     // Min Cone Angle
    m_Properties.Segment_Deviation_Max_Cone = 30.0f;    // Max Cone angle
    m_Properties.Segment_Deviation_Rotate   = 10.0f;    // Max Polar rotation

    m_Properties.Length_Falloff_Scale       = 0.1f;     // Segment Length falloff
                                                        // happens 10 times slower
                                                        // than branch thickness
                                                        // fall off

    // Deviation properties for new child branch
    m_Properties.Split_Deviation_Min_Cone   = 10.0f;    // Min Cone Angle
    m_Properties.Split_Deviation_Max_Cone   = 70.0f;    // Max Cone Angle
    m_Properties.Split_Deviation_Rotate     = 360.0f;   // Max Polar rotation

    m_Properties.SegDev_Parent_Weight       = 0.0f;     // Weight with which the
                                                        // original direction of
                                                        // the segments parent is
                                                        // averaged with the
                                                        // deviation

    m_Properties.SegDev_GrowthDir_Weight    = 0.4f;     // Weight with which the
                                                        // growth direction vector
                                                        // is averaged with the
                                                        // deviated segment
                                                        // direction

    m_Properties.Branch_Resolution          = 8;        // Number of vertices used
                                                        // for each branch mesh
                                                        // segment ring

    m_Properties.Bone_Resolution            = 3;        // One bone every 3 nodes

    m_Properties.Texture_Scale_U            = 1.0f;
    m_Properties.Texture_Scale_V            = 1.0f;
```

```
    // Growth direction of tree
    m_Properties.Growth_Dir                 = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );

    // Setup default material properties
    m_Material.Diffuse   = D3DXCOLOR( 0.8f, 0.8f, 0.8f, 0.8f );
    m_Material.Emissive  = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 1.0f );
    m_Material.Ambient   = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 1.0f );
    m_Material.Specular  = D3DXCOLOR( 0.0f, 0.0f, 0.0f, 1.0f );
    m_Material.Power     = 0.0f;

}
```

What is important to note is that the first thing we do in this constructor is issue a call to the base class constructor so that CActor can perform its own initialization. Remember, the base class sets a default skinning method and initializes the actor's callback array. It also sets some default parameters for the animation controller.

Destructor – CTreeActor

Like all of our destructors, this destructor is also very simple due to the fact that it hands off the clean-up to its Release method (which can also be called by the application to force a clean up). What we do release in this function is any texture filename string that may exist.

When we look at the CTreeActor::SetBranchMaterial method in a moment, you will see that the texture filename passed in is duplicated and stored in the m_strTexture member variable (using _tcsdup). Therefore, we must make sure on destruction that we clean that memory up in the appropriate way. We then issue a call to the CTreeActor::Release method to do any final clean up.

```
CTreeActor::~CTreeActor()
{
    // Release any allocated memory
    Release();

    // Release the properties.
    if ( m_strTexture ) free( m_strTexture ); // Allocated by _tcsdup

    // Reset variables
    m_strTexture = NULL;
}
```

Release – CTreeActor

The Release method is small because the derived class has very few resources to clean up. What we must do however is call the base class Release method so that the frame hierarchy and meshes managed by the base class are released. We know the CActor base class performs quite a bit of clean up, such as releasing interfaces to animation controllers, destroying CTriMesh objects, and de-allocating the entire

frame hierarchy. Once we have called the base class's Release method, the only additional information that needs to be released in CTreeActor is the virtual tree hierarchy. When we looked at the BranchNode structure earlier, we also saw how its constructor took care of deleting its sibling and child pointers. Therefore, all we have to do is delete the root branch node (m_pHeadNode), which will delete its child, which will delete its child, and so on, causing a traversal and deletion of the entire branch node hierarchy.

```
void CTreeActor::Release()
{
    // Call base release
    CActor::Release();

    // Destroy the tree hierarchy data
    if ( m_pHeadNode  ) delete m_pHeadNode;

    // Clear variables.
    m_pHeadNode  = NULL;
}
```

SetGrowthProperties / GetGrowthProperties – CTreeActor

One task that the application will want to perform before generating the actual tree, is setting the growth properties that will influence the tree generation process. This function takes as its only parameter a TreeGrowthProperties structure whose values will be stored in the CTreeActor's m_Properties member. We also supply a function for retrieving the properties of a CTreeActor. Both the 'Set' and 'Get' functions are shown below.

```
void CTreeActor::SetGrowthProperties( const TreeGrowthProperties & Prop )
{
    // Store the properties
    m_Properties = Prop;
}
```

```
TreeGrowthProperties CTreeActor::GetGrowthProperties( ) const
{
    // Return the properties
    return m_Properties;
}
```

It is important that you set the growth properties of the tree prior to calling the GenerateTree method since it uses the values stored in the m_Properties structure to influence tree generation. Setting the properties after the tree has been generated will have no effect.

SetBranchMaterial – CTreeActor

Another task that you will probably want to perform before calling GenerateTree is informing the tree about the texture and material you would like to have applied to its mesh faces. We do this using the CTreeActor::SetBranchMaterial function. The function accepts two parameters. The first should be a string containing the filename of the texture you wish to use and the second parameter should be a pointer to a D3DMATERIAL9 structure containing the light reflectance properties that should be applied to the faces of the tree.

```
void CTreeActor::SetBranchMaterial( LPCTSTR strTexture, D3DMATERIAL9 * pMaterial )
{
    // Free any previous texture name
    if ( m_strTexture ) free( m_strTexture );
    m_strTexture = NULL;

    // Store the material
    if ( pMaterial ) m_Material = *pMaterial;

    // Duplicate the texture filename if any
    if ( strTexture ) m_strTexture = _tcsdup( strTexture );
}
```

If the tree already has a texture name set, then its memory is released. The material passed is then copied into the m_Material member variable, replacing the default values we assigned to it in the constructor. Finally, we use the _tcsdup function to make a copy of the texture filename. A pointer to this copy is stored in the m_strTexture member variable.

GenerateTree - CTreeActor

We are now ready to examine the GenerateTree method, which is invoked by the caller to build the entire tree. We can think of this method as being the replacement to the regular actor's LoadActorFromX method in that on function return, the actor will have a fully populated frame hierarchy (and potentially even some animation data). This function is actually just a front end function for the caller since the tree building mechanism is mostly done through helper functions called from this method. We will examine each of the helper functions one at a time afterwards. However, by looking at the GenerateTree function first, we will be able to see the overall order in which the various processes are invoked.

GenerateTree accepts five parameters (the final two are optional). The first parameter is a combination of one or more D3DXMESH flags which instruct the tree in which of the device's resource pools the vertex and index buffers for the branch meshes should be allocated. The second parameter is a pointer to the device that will essentially own the actor and its meshes. The third parameter is the initial dimensions for the root node in the branch node hierarchy. For example, passing a vector of (2, 3, 6) means the thickness of the root branch segment is defined as an ellipse that has radii X=2 and Y=3 in world space. These values are used to place the ring of vertices at the root branch node of the correct

size. The size of each branch segment will get smaller as we step along the branch. That is why we only pass in the size of the root branch node which will then be automatically downsized in each recursive step. The Z component of this vector describes the length of the root branch segment. Using the vector specified above, this means after the root node has been inserted, we will move a distance of 6 units along the root node's direction vector before placing the next node in that branch. As both these nodes define the bottom and top of the same cylinder, this Z value really defines the length of the root branch segment. The fourth (optional) parameter is another 3D vector describing the initial direction vector of the root node of the virtual tree. If we were to pass a vector of <1,0,0> the initial segment of the root branch would grow horizontally along the X axis (e.g., out of a cliff face). If the growth vector of the tree has been set (in the tree properties) to <0,1,0>, the tree would start to grow upwards as each new node was added. If you omit this parameter, a default vector of <0,1,0> will be used. This is a very sensible default since you will usually want your tree to grow vertically upwards. The final (optional) parameter is an unsigned long integer that will be used to seed the random number generator. If omitted, the current system time will be used to seed the random number generator.

In the first section of the function, we allocate an instance of our CAllocateHierarchy class. This will be used later so that we can call its CreateFrame and CreateMeshContainer methods to generate the frames of the hierarchy and the skinned meshes. We return if a valid device was not passed.

```
HRESULT CTreeActor::GenerateTree( ULONG Options, LPDIRECT3DDEVICE9 pD3DDevice,
                                  const D3DXVECTOR3 & vecDimensions,
                                  const D3DXVECTOR3 & vecInitialDir ,
                                  ULONG BranchSeed )
{
    HRESULT            hRet;
    CAllocateHierarchy Allocator( this );

    // Validate parameters
    if ( !pD3DDevice ) return D3DERR_INVALIDCALL;

    // Release previous data.
    Release();

    // Store the D3D Device here
    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();

    // Store options
    m_nOptions = Options;
```

Provided the device is valid we call the CTreeActor::Release method to release any previous data that may have been generated for this tree object. This allows us to use the same object to generate another tree simply by calling its GenerateTree method again. Remember, the Release method also passes the request to the base class Release method, ensuring that all frames, mesh, and attribute data is released before generating a new tree from scratch. Next we make a copy of the device interface, increment the interface's reference count, and store the mesh creation options in the m_nOptions member.

The final section of the function code builds the entire tree through the invocation of three other methods, each of which is responsible for one phase of the tree generation. We will look at the code to each of these three functions as we go along.

```
    // Generate the branches
    hRet = GenerateBranches( vecDimensions, vecInitialDir, BranchSeed );
    if (FAILED(hRet)) return hRet;

    // Build the frame hierarchy
    hRet = BuildFrameHierarchy( &Allocator );
    if (FAILED(hRet)) return hRet;

    // Build the bone matrix tables for all skinned meshes stored here
    if ( m_pFrameRoot )
    {
        hRet = BuildBoneMatrixPointers( m_pFrameRoot );
        if ( FAILED(hRet) ) return hRet;

    } // End if no hierarchy

    // All is well.
    return D3D_OK;
}
```

The GenerateBranches function is called first and completes the first phase of the tree building process. It is responsible to building the branch node hierarchy that contains our virtual tree representation. When the GenerateBranches method returns, we will have a virtual tree hierarchy, the root branch node of which will be pointed to by the m_pHeadNode member variable. At this point, the actor will not yet have had its frame hierarchy built and no mesh data will have been generated. That is what the next function BuildFrameHierarchy does. It accomplish phase two of the tree generation process.

BuildFrameHierarchy is responsible for building the actor's D3DXFRAME hierarchy and generating all skinned meshes. It does so by traversing the virtual tree hierarchy assembled in phase one and inserting vertices and frames for the relevant branch nodes. Once the BuildFrameHierarchy function returns, the actor's frame hierarchy will have been created and all the skinned meshes will have been created and attached to that hierarchy as well. However, at this point there is still one final step to perform which you should recall from the last chapter. During the building of the frame hierarchy, we cannot store the bone pointers needed for our mesh containers until the frame hierarchy has been created in its entirety. Therefore, just as we do in the CActor::LoadActorFromX function, after the frame hierarchy has been created, we traverse it and calculate the absolute matrices at each frame (the bone matrices) before storing all the bone matrices used by a given mesh container in its bone matrix pointer array. We looked at the code to the CActor::BuildBoneMatrixPointers function in the previous chapter's workbook.

Now we are ready to look at the first function called in the above code which implements phase one (i.e., the building of the virtual tree branch node hierarchy).

GenerateBranches - CTreeActor

This function is called from the GenerateTree method and is passed the dimensions of the root branch node, the direction of the root branch node, and (optionally) the random seed. The first thing the function does is seed the random number generator. The default value of the Seed parameter is zero if omitted from the parameter list, so the first thing we do if this is the case is set the seed value to the value returned from the timeGetTime function. We then store either the passed seed value or the seed value just calculated in the m_nBranchSeed member variable. Finally, we seed the random number generator via a call to srand.

```
HRESULT CTreeActor::GenerateBranches( const D3DXVECTOR3 & vecDimensions,
                                      const D3DXVECTOR3 & vecInitialDir ,
                                      ULONG Seed )
{
    ULONG i;
    ULONG BranchUID = 0;

    // Seed the random number generator with current time if not specified
    if ( Seed == 0 ) Seed = timeGetTime();

    // Store the seed value
    m_nBranchSeed = Seed;

    // Seed the generator
    srand( Seed );
```

This function is responsible for creating the root node of the branch node hierarchy. It will then pass this node to the BuildBranchNodes function (a recursive function that will repeatedly call itself to create new branch nodes for the hierarchy). However, as mentioned earlier, we do have the option of creating a tree that has multiple initial branches (multiple trunks), so we may have multiple nodes in the root level of the tree. While we will usually expect a value of 1 to be stored in the Initial_Branch_Count member of the tree's growth properties structure, this may not always be the case. Therefore, we will set up a loop that counts up to the value stored in the Initial_Branch_Count member and create a new branch node for each root branch. As we may be creating multiple branch nodes at the root level (one for each initial branch) we will connect these as siblings at the root level. Each root branch node will be passed into the BuildBranchNodes recursive function to generate the entire tree for that branch recursively.

In the next section of code we show the start of that loop. Each iteration allocates a new branch node. If a previous node has been created in a previous iteration then we attach the new branch node to the previous node's sibling pointer. If not, then this is the first node we have generated and will assign the m_pHeadNode member variable to point at this node. In other words, if the initial branch count was set to 3, in the first iteration we would generate a new branch node and assign it to the m_pHeadNode pointer. In the second iteration we would create a new branch node and assign it to the head node's sibling pointer. In the third and final iteration we would create a new branch node and assign it to the sibling pointer of the node we created in loop iteration 2, and so on. Thus, we are creating a sibling list of Branch_Begin nodes at the root level of the hierarchy.

```
    BranchNode * pPrevNode = NULL;

    // Generate the required set of head branches
    for ( i = 0; i < m_Properties.Initial_Branch_Count; ++i )
    {
        BranchNode * pNewNode = new BranchNode;
        if ( !pNewNode ) continue;

        // Store in the appropriate place
        if ( pPrevNode )
            pPrevNode->Sibling = pNewNode;
        else
            m_pHeadNode = pNewNode;

        // Setup the node
        pNewNode->UID        = BranchUID++;
        pNewNode->Type       = BRANCH_BEGIN;
        pNewNode->Direction  = vecInitialDir;
        pNewNode->Dimensions = vecDimensions;
        pNewNode->Iteration  = 0;
```

In the above code, you can see that for each root node we create we assign it a unique branch node ID (BranchUID). This value was set to 0 at the start of this function and is incremented with each new node we add. We also know that any branch nodes we create at the root level will be the start nodes of trunk branches, so we set the type of each branch node created in this loop to the type Branch_Begin. When building the meshes later, we will know that each of these nodes starts a new branch mesh. We also store the direction vector and the dimension vector passed into the function in the node. Finally, we set the iteration of the nodes to zero. This iteration will be passed down the branches and incremented at each level. Every branch node added at the root level however will be set to zero since they exist at the same level in the hierarchy and for all intents and purposes are the start nodes of separate trees linked at the root level. The position of each node is not set here as this will have been set to 0 in the constructor of the branch node structure. We wish each trunk branch to be positioned at (0,0,0) in tree space.

We know the direction vector of the nodes we are creating because it was passed into the function. However, we learned earlier that each node also needs a right vector. The right vector of the root node(s) will be passed down the tree and modified at each node so that it remains correctly orthogonal to the node's deviated direction vector. We do not require the application to pass in the right vector of the root node(s) as we can calculate that easily. All we have to do is find the coordinate system axis (X, Y or Z) that is least aligned to the direction vector. This will allow us to find a vector that is definitely not the same as the direction vector. We find this axis simply by taking the absolute values of the direction vector's components and searching for the smallest component. For example, we know that if the direction vector has a Z component of 0, then we can use the Z axis of the coordinate system since this is definitely not close to being aligned to the direction vector. All we are doing here is safely picking an axis which is not the direction vector. Once we have that axis vector, we can perform the cross product between the direction vector and that axis vector to create the right vector for the node. The following code shows how we find this axis vector and cross it with the direction vector to generate the right vector. The right vector is then stored in the branch node.

```
        // Get absolute normal vector
        float x = fabsf( pNewNode->Direction.x );
```

```
        float y = fabsf( pNewNode->Direction.y );
        float z = fabsf( pNewNode->Direction.z );
        float fNorm = x;

        // Find the best vector to use as right vector
        D3DXVECTOR3 vecCross = D3DXVECTOR3( 1.0f, 0.0f, 0.0f );
        if ( fNorm > y )
       { fNorm = y; vecCross = D3DXVECTOR3( 0.0f, 1.0f, 0.0f ); }

        if ( fNorm > z )
       { fNorm = z; vecCross = D3DXVECTOR3( 0.0f, 0.0f, 1.0f ); }

        D3DXVec3Cross( &pNewNode->Right, &pNewNode->Direction, &vecCross );
```

At this point we have successfully populated the new branch node with its information. However, if we are generating multiple branches at the root level (Initial_Branch_Count > 1) then we cannot simply assign the same direction and right vectors to each branch node; otherwise, each root branch node we create will be placed in the exact same position and with the same orientation as the other nodes in its sibling list. So, if we are only creating a single node, then our job is done and we pass this node into the BuildBranchNodes function to start the recursive branch building process. The branch node will have the exact direction vector that we passed into the function. However, if we are creating multiple root branch nodes, then we must deviate the direction vector of each node so that they are slightly different. This will allow us to generate a tree where multiple trunks sprout from the same location but with different directions. If deviation is needed, we call the DeviateNode function to perform that deviation. This function (which will be covered in a moment) is passed the node, the min and max cone angles and a rotation angle. This will deviate the vector a random angle about its right vector with a range of –180 to +180 degrees. It will also rotate the new deviated direction vector a random angle of 180 degrees about the original direction vector. Finally, it will rotate the node's right vector so that it stays orthogonal to the new deviated direction vector.

We covered how vector deviation is performed in theory earlier on in the chapter, so refer back if you need to brush up. After the DeviateNode function returns, we have modified the root node direction and right vectors and are ready to pass this node into the BuildBranchNodes function. This function is a recursive process that repeats until all the branch segments of the root node and any child branches it spawns are calculated. Suffice to say, when BuildBranchNodes returns program flow back to this function, the entire virtual tree for the current root node we are processing will have been created.

```
        // Deviate if we're starting with more than one branch
        if ( m_Properties.Initial_Branch_Count > 1 )
             DeviateNode( pNewNode, 0.0f, 180.0f, 180.0f );

        // Start the recursive build for this branch
        BuildBranchNodes( pNewNode, BranchUID );

        // Store previous node
        pPrevNode = pNewNode;

    } // Next Initial Branch
```

At the end of the loop shown in the above code, we assign the pPrevNode pointer to point at the node we have just generated. This way, in the next iteration (assuming multiple roots) of the loop, we have access to the node generated in the previous iteration so that we can add a new root node to its sibling list.

When the loop finishes, the entire virtual tree hierarchy will have been created. We will examine the BuildBranchNodes recursive function in detail shortly, as this is where all of the work happens for phase one.

In the next and final section of the code we test to see that the root node of the tree is valid and if not we return E_OUTOFMEMORY. If this is NULL, then the very first root node could not be allocated so we must return failure. Otherwise, we return success.

```
    // Fail if we failed.
    if ( !m_pHeadNode ) return E_OUTOFMEMORY;

    // Success!!
    return D3D_OK;
}
```

There are three functions called from the previous function that we need to discuss. The first one called is DeviateNode. It will be used many times throughout the tree building process to randomly deviate the direction vector of a child node from that of its parent node. Let us have a look at that function next.

DeviateNode - CTreeActor

DeviateNode is a small function that is called many times throughout the tree generation process to randomly deviate the vector of a child node from that of its parent over some given a range. It takes four parameters. The first parameter is the node whose direction vector we wish to deviate. The direction vector of a newly created node that is passed into this function will initially have its direction vector inherited from the parent node (or in the case of the GenerateBranches function, explicitly set to the direction vector passed by the application). Therefore, there is no reason to also pass in the parent node's vectors (direction and right) which we wish to rotate around, as initially, the vectors stored in the child node we pass will be equal to its parent vectors.

The second and third parameters specify the minimum and maximum deviation angles in degrees. This is the amount we wish to rotate the direction vector clockwise or counter-counter clockwise about the parent node's right vector. This forms a minimum and maximum cone of rotation. We can initially perform this rotation of the direction vector about the node's right vector, since at this point the right vector of the node will be equal to the right vector stored in its parent node. The final parameter is the maximum polar rotation angle we wish to rotate the (now rotated) direction vector around the parent's direction vector. Passing a value of 180 degrees will allow us to swing the direction vector to any position around the branch (90 degrees left or right from the rotated position). Here is the first section of the code followed by an explanation.

```
void CTreeActor::DeviateNode( BranchNode * pNode,
                              float fAzimuthThetaMin,
                              float fAzimuthThetaMax,
                              float fPolarTheta ) const
{
    // Cache direction
    D3DXVECTOR3 vecNormal = pNode->Direction;

    // Generate Azimuth / Polar angles
    float fAzimuth = (float)rand() / (float)RAND_MAX;
    float fPolar   = (float)rand() / (float)RAND_MAX;
    fAzimuth = fAzimuthThetaMin +
               ((fAzimuthThetaMax - fAzimuthThetaMin) * fAzimuth);

    fPolar   = (fPolarTheta * fPolar) - (fPolarTheta / 2.0f);
```

First we make a copy of the node's direction vector. This is important as this currently holds the direction vector of the parent node. Once we rotate this vector it will be overwritten and we will have lost our ability to rotate the rotated direction vector about the original direction vector of the parent.

We then generate two random rotation values for the fAzimuth and fPolar local variables. The rand function generates a random integer in the range [0, RAND_MAX], so dividing the result of rand by RAND_MAX gives a random float in the range [0.0, 1.0]. We calculate the fAzimuth angle using the [0.0, 1.0] random number to find the angle between the minimum cone angle and the maximum cone angle using interpolation. That is, to calculate the first rotation angle (around the parent's right vector), we the minimum cone angle with the difference between the min and max cone angles multiplied by our random float.

We now have an angle in degrees between the minimum and the maximum cone angle. Next we calculate the polar rotation angle. The 0.0 to 1.0 range random number represents an angle between 0 and the maximum polar rotation angle passed into the function. Therefore, scaling the maximum polar rotation angle by the random float will produce an angle value in degrees, in the range [0.0, Max Polar Rotation Angle]. If the passed polar rotation limit was 180 degrees for example, this means we wish to map it into the [–90, +90] range so that we have a random chance of rotating the direction vector clockwise or counter clockwise. Therefore, we subtract from the angle the maximum angle passed divided by two. This maps a value of (Polar Angle / 2) to a zero degree rotation, a value of 0 to –(Polar Angle/2) and a value of Polar Angle to (Polar Angle / 2). This is exactly what we want.

Now that we have our two rotation angles, we need to decide whether the first rotation angle (fAzimuth) is going to rotate the direction vector of the node clockwise or counter-clockwise around the parent's right vector. The following code uses a member function called ChanceResult to generate a number between 1 and 100. This function returns true if it generated a random number smaller than the value passed in (typically a value in the range of 0 to 100 also). We use this function frequently through the tree generation process to decide whether a branch node should be deviated or whether it should spawn child branches. Because we want an equal probability for each direction, we pass in a value of 50. The function will return true if it generates a random number between 0 – 49, otherwise, it will return false. As you can see, the parameter we pass into this function is really a probability value; the lower the value we pass in, the less chance the function has of returning true.

```
// Deviate in both directions to prevent a tendency to lean, whilst still
    // providing support for our 'dead zone' of rotation.
    if ( ChanceResult( 50 ) == true ) fAzimuth = -fAzimuth;
```

If the function returns true, we negate the rotation angle so that rotation happens in the opposite direction.

Now we have two final rotation angles, and it is time to perform the rotations. In the first step, we build a rotation matrix that will rotate vectors about the node's right vector (currently equal to the parent's right vector) by fAzimuth degrees. We then rotate the node's direction vector about the right vector by transforming it using this matrix. This step rotates the local Z axis of the new node into either the left or right semi-circle surround the parent node direction vector.

```
    // Rotate the normal
    D3DXMATRIX mtxRot;
    D3DXMatrixRotationAxis( &mtxRot, &pNode->Right, D3DXToRadian( fAzimuth ) );
    D3DXVec3TransformNormal( &pNode->Direction, &vecNormal, &mtxRot );
```

The first rotation has been performed and our node's direction vector will now be rotated either clockwise or counter-clockwise about the parent's right vector. Next we want to perform the polar rotation. In order to do this we must rotate the node's direction vector around the parent's direction vector (which is why we made a copy in vecNormal at the start of the function) by an angle of fPolar degrees. We build a matrix that performs this rotation and multiply both the node's direction vector and right vector by this matrix. This rotates the direction vector into its final position and also rotates the right vector by the same amount such that it remains orthogonal to the direction vector.

```
    D3DXMatrixRotationAxis( &mtxRot, &vecNormal, D3DXToRadian( fPolar ) );
    D3DXVec3TransformNormal( &pNode->Direction, &pNode->Direction, &mtxRot );
    D3DXVec3TransformNormal( &pNode->Right, &pNode->Right, &mtxRot );
}
```

Finally, we run some vector generation code to ensure that the right vector and direction vector are completely orthogonal. We discussed a similar technique in Chapter Four of Module I when dealing with vector re-generation for the camera class. Because we are passing a direction and right vector down through the recursive process and are repeatedly applying rotations to it, without this next step, floating point accumulation errors would build quite quickly and we would lose orthogonality. Since these vectors represent our node coordinate system which is used in many places during tree and mesh creation, we must not let this happen.

```
   // Ensure that these new vectors are orthogonal
   D3DXVec3Cross( &vecNormal, &pNode->Direction, &pNode->Right );
   D3DXVec3Cross( &pNode->Right, &vecNormal, &pNode->Direction );
   D3DXVec3Normalize( &pNode->Right, &pNode->Right );
```

ChanceResult - CTreeActor

Before continuing to cover the other functions called by GenerateBranches, let us have a look at the ChanceResult function which was called from DeviateNode (and called from various other places throughout the tree generation process).

The function generates a value between 0.0 and 1.0 by dividing the value returned from rand by RAND_MAX. We then multiply this float by 100 so that we have a value between 0.0 and 100.0. If this value is smaller than the probability value passed in (itself a value between 0.0 and 100.0) the function returns true (simulating that the probability has come true in this instance). Otherwise, false is returned.

```
bool CTreeActor::ChanceResult( float fValue ) const
{
    // REALLY simple percentage chance prediction
    if ( (((float)rand() / (float)RAND_MAX) * 100.0f) < fValue ) return true;
    return false;
}
```

BuildBranchNodes – CTreeActor

Recall that after GenerateBranches has deviated the direction vector for each root node (in the multiple root branch case) it passes the root node into the BuildBranchNodes function. This function is called once by GenerateBranches for each root node. The BuildBranchNodes function is really the heart of the tree generation process. Not surprisingly, it is a recursive function. When the initial call made from GenerateBranches returns, the entire branch node hierarchy for that root branch will have been created.

While recursive functions are often tricky to follow along with in your head, the tasks that this function must perform are really quite simple. The function is passed a branch node that has already been generated and it must create a new child node for that node. It must also decide whether the new child node (which continues the branch) should be a normal branch node or whether it should be a Branch_End node. It must also decide whether the child node it has generated will spawn additional child branches (new Branch_Begin nodes). Of course, when making these decisions, it will use the growth properties structure that the application passed in when starting the process.

When the function creates and attaches the new child node, it fills out all its information and then sends that new node into the BuildBranchNodes function with a recursive call. Therefore, each call to this function will generate a new node, attach it to the parent node (i.e., the node passed in) and recursively call itself passing in the new node as the parent node in the next recursive call. This process continues until an instance of the function determines the end of the branch as been reached and returns.

Recall from our discussion of GenerateBranches that this function was passed two parameters. The first was a root node of a root branch and the second was the current value of the BranchUID variable. The BranchUID variable was local to the GenerateBranches function and was incremented every time a root node was created. However, we want this value to be incremented for every node that is created, so we

pass this value by reference into the BuildBranchNodes call. This means that with each recursive call to BuildBranchNodes when a new branch node is created and attached to the tree, the same physical variable is accessed from each level of the recursion. The value will be increased with every node created and then assigned to that node as its unique ID.

What was not obvious in the GenerateBranches function is that BuildBranchNodes function accepts a third parameter: the current iteration count. Recall that in each node we store the current iteration which is incremented with each recursive call. This value represents the current depth of the tree where the node is stored. We use this iteration count for both determining when a branch has become too long (and should be terminated), and when calculating the texture coordinates for the branch meshes. We saw in the GenerateBranches function that each root node was assigned a value of 0. This is as we might expect because the nodes in the root level are in the first level of the hierarchy. However, that function did not pass in the iteration parameter when it called BuildBranchNodes for each root generated, which means the default iteration value of 0 will be used when adding the second node of each root branch to the tree. The result is that both the first and second nodes in a branch will have an iteration count of zero. For all nodes we add to the branch after that, it is incremented. Thus, the third branch segment will have an iteration count of 1 and the fourth will have an iteration count of 2, and so on. (it will always be incremented for all recursive calls from that point on). Is this a mistake? Not at all!

Although we have not discussed this concept previously, we do this because it is more intuitive during the tree building process if the iteration count of a branch node actually describes the index of the branch segment (cylinder) it creates. When we add the initial node, no segment is created because we need two nodes to make a segment. When the second node is added to the branch it is assigned an index count of 0 since its addition to the branch essentially adds the first segment to the branch. When we add the third node to the tree (node 2 because it is zero based), this is assigned an iteration count of 1 since its addition to the branch adds the second segment (segment 1), and so on.

Let us have a look at the code a couple of sections at a time.

```
void CTreeActor::BuildBranchNodes( BranchNode * pNode,
                                   ULONG & BranchUID,
                                   ULONG Iteration /* = 0 */ )
{
    ULONG NewNodeCount = 1, i;
    bool bEnd = false;

    // Bail if this is a branch end
    if ( pNode->Type == BRANCH_END ) return;
```

In the first section of code we set a local variable called NewNodeCount to 1. Keep in mind that, assuming the parent node we have been passed is not a branch end node, we will always add at least one node to this branch (even if it is just a Branch_End node that will cause the processing of this branch to terminate in the next recursive call). The reason we use this variable is due to the fact that we may increment this number throughout the body of the function if it is determined that the passed parent node should not only spawn another branch segment, but also one or more new branch start segments. If we decide later in the function that 2 new branches are going to be spawned from this node, this value will be set to 3. We can then create a loop that creates these three new nodes as siblings and attaches them to the parent node's (the node passed in) child list.

Notice how we initially set a boolean variable called bEnd to false. Provided the parent node is not already a branch end node, we will continue the branch by at least one more node. If at some point we determine that the child node we create should end the branch, we will set this boolean to true. This will be used to setup a branch end node instead of a normal branch node and ensure that when the next recursive call happens, and the parent branch node passed in is an end node, the recursive path will terminate all the way back to the last un-processed fork along the recursive road.

We can see this termination process being performed at the bottom of the code shown above. It tests the parent node to see if it is of type BRANCH_END. If it is, then the branch has ended and we should not add any more branch segments to that branch. In this case, we simply return. This terminates the recursive process of adding nodes to that branch.

If program flow gets past the code shown above, then it means the parent node passed into the function is not an end node and we need to continue this branch by at least one more node. What we must determine however is if this new node we are about to add to the branch is an end node. We perform a series of tests to determine this, and if any of these tests pass, the bEnd local boolean is set to true so that we know the new node we are about to create and add to this branch should be an end node.

First we test if the iteration value passed into the function (which is incremented every time the function is called and we step down another level in the hierarchy) is equal to the Max_Iteration_Count member of the tree's growth properties. This is a fixed termination level and our way of limiting the depth of the hierarchy that is ultimately created. If this is the case, then the recursive process has generated a tree that is of the maximum depth and the next branch node we add should be a branch end node. Thus, we set the branch end boolean to true, as shown below. Remember, the Iteration variable is a parameter to the function that is increased as the function repeatedly calls itself to step along the branch.

```
    // We've reached our end point here.
    if ( Iteration == m_Properties.Max_Iteration_Count ) bEnd = true;
```

Next we test to see if there is any chance of the node we are about to create spawning new child branches. In the TreeGrowthProperties structure we have several members that help us control the circumstances under which splits in the branch might happen.

First, we will definitely not want to create any splits if the iteration count passed is already larger than the maximum iteration count set for the tree. Otherwise, we would be generating a new branch at a level in the hierarchy that is deeper than we want the hierarchy to be. Therefore, we will certainly only consider splitting from this node if the current iteration count (tree depth) is smaller than the maximum iteration count set for the tree. Additionally, we only introduce new child branches at this node if the X and Y dimensions of the parent node (which are tapered as we step along the branch) are within the size ranges specified by the Min_Split_Size and Max_Split_Size tree growth properties. If either the X or Y dimensions fall outside this range, then the thickness of the branch is currently such that the application does not wish splits to happen. These properties are great for making sure that we do not spawn branches from very slim parent branches or that we do not spawn them from the base of a trunk branch where the branch is the thickest. The following code shows the conditional and the code block that will be executed only if we are allowed to spawn child branches from this node.

```
    // Chance of splitting into N ?
    if (  Iteration <  m_Properties.Max_Iteration_Count &&
          Iteration >= m_Properties.Min_Split_Iteration &&
          Iteration <= m_Properties.Max_Split_Iteration &&
          pNode->Dimensions.x >= m_Properties.Min_Split_Size &&
          pNode->Dimensions.x <= m_Properties.Max_Split_Size &&
          pNode->Dimensions.y >= m_Properties.Min_Split_Size &&
          pNode->Dimensions.y <= m_Properties.Max_Split_Size )
    {
        if ( ChanceResult( m_Properties.Two_Split_Chance ) )   NewNodeCount = 2;
        if ( ChanceResult( m_Properties.Three_Split_Chance ) ) NewNodeCount = 3;
        if ( ChanceResult( m_Properties.Four_Split_Chance ) )  NewNodeCount = 4;

        // Are we splitting here
        if ( NewNodeCount > 1 )
        {
            // Chance that a split will terminate this branch.
            if ( ChanceResult( m_Properties.Split_End_Chance ) ) bEnd = true;

        } // End if split here

    } // End if we've reached our limit
```

So what is happening inside the code block above? For starters, the fact that we are in the code block means that a new branch *can be* spawned from this node. Of course, it does not mean a new branch *will be* spawned since this will be decided using the split probability values stored in the tree growth properties structure.

Notice that we make three calls to the ChanceResult function (covered earlier). First we call it for the two split case. The chances of this test succeeding depend on the value stored in the Two_Split_Chance tree growth property. The higher the probability value we pass in, the better the odds that it will return true. We perform the two split, three split, and four split tests in that order such that higher branch splits (which typically have much lower probabilities assigned by the application) take precedence. As shown above, at the end of these tests, the NewNodeCount local variable will have been updated to contain the exact number of nodes that should be created at this level of the branch and attached to the parent node passed in as child nodes. This number is the sum of the new child branches we wish to add, plus one for the node that continues the current branch.

In the final section of the code above, you can see that provided we have introduced at least one new child branch at this node (NewNodeCount > 1), we will perform another probability test to see if the normal branch node we are about to add to the parent node (to continue the current branch) should be an end node terminating the current branch being processed. The Split_End_Chance growth probability variable is used for this test. This allows the application to control the likelihood that a branch will terminate at a node where new child branches are spawned (i.e., a fork in the branch).

At this point in the function, we know we have to create at least one new node to continue the branch, even if that node is an end node. In addition, we may also have to create one Branch_Begin nodes for each new branch starting at this new node. Forgetting about the new child branch nodes for now, let us first create a node that will be the continuation of the branch we are currently processing.

Before we create our new child branch node, we must calculate the size of this node. As discussed previously, every node in a branch will be smaller than the node before it (the parent node) so that our branches gradually taper off into tips by the time we reach the end node of that branch. In the next step we will calculate the dimensions of the new node we are about to create. How do we calculate the size of each node so that we have a gradual tapering from segment to segment such that the dimensions of the end node of the branch are zero?

We know that the dimensions of the root node describe the thickest part of the tree. That is, the X and Y dimensions of the root node describe the largest ring of vertices we will create. We also know that the TreeGrowthProperties::Max_Iteration_Count member contains the maximum depth of the hierarchy. Essentially, this tells us the iteration where the branch should have zero dimensions. Therefore, if we divide the X and Y dimensions of the root node by the maximum iteration count, we have a value that we can subtract from each node during each iteration. For example, if the root X dimension was 100 and the maximum iteration count was 25, we would calculate the amount to subtract from the dimensions of each node as follows:

SubtractAmount = 20 / 5 = 4;

So we would need to subtract 4 in each recursion from the X dimensions of the parent node when calculating the child node's dimensions. Remembering that in the next iteration, the child node will be the parent that will be reduced by another 4 units when calculating the dimensions of its child, we can see that the X dimension at each recursion would be:

Node 0 : X = 20
Node 1 : X = 16
Node 2 : X = 12
Node 3 : X = 8
Node 4 : X = 4
Node 5 : X = 0 (end node with single vertex)

As you can see, using this calculation we can recursively reduce the size of the nodes in each level of the hierarchy such that at the maximum iteration count, we have a branch tip node with zero size. We do this for both the X and Y dimensions since they may have different radii. We also do the same with the Z dimension since we want the length of each segment to get smaller as we traverse the length of the branch. However, we normally do not want the same level of downscaling from node to node as this would create very small stumpy segments at the ends of the branches. Unlike the X and Y dimensions, we are not aiming to get the length of the branch segment to be zero at the end node. To control the falloff in Z, we introduce an additional property (Length_Falloff_Scale) which can be used to allow the reduction in length from node to node to happen at a reduced rate. For example, if we set this property to 0.25, the length of the nodes will get smaller at a ratio of ¼ of the size reduction in branch thickness.

This next section of code calculates the three reduction values we need to subtract from the parent node's dimensions in order to calculate the dimensions of the new node.

```
    // Scale Ratio
    float ScalarX = (m_pHeadNode->Dimensions.x /
                    (float)m_Properties.Max_Iteration_Count);
```

```
    float ScalarY = (m_pHeadNode->Dimensions.y /
                    (float)m_Properties.Max_Iteration_Count);

    float ScalarZ = ((m_pHeadNode->Dimensions.z /
                    (float)m_Properties.Max_Iteration_Count) *
                     m_Properties.Length_Falloff_Scale );

    // Do we have enough room for another segment?
    if ( pNode->Dimensions.x < ScalarX) bEnd = true;
    if ( pNode->Dimensions.y < ScalarY) bEnd = true;
    if ( pNode->Dimensions.z < ScalarZ) bEnd = true;
```

As the above code demonstrates, after we have calculated the three reduction amounts (ScalarX, ScalarY and ScalarZ) we next test to see if any of these values are larger than the current dimensions of the parent node. If this is the case then it means the parent node is either too thin or too short to reduce by a suitable amount. When this happens, the child node we are about to add should end the branch (notice that we set the bEnd boolean to true).

At this point, we are ready to create the new child node that continues the current branch. We will deal with any new branch start nodes we may have decided to create in a moment. All we are dealing with here is the node that will continue the branch to which the parent node belongs. Here is the code that allocates a new child branch node and populates its members with the correct information.

```
    // Generate segment node (continuation of the branch)
    BranchNode * pNewNode  = new BranchNode;

    if ( !pNewNode ) return;

    // Store node details
    pNewNode->UID          = BranchUID++;
    pNewNode->Parent       = pNode;
    pNewNode->Dimensions.x = pNode->Dimensions.x - ScalarX;
    pNewNode->Dimensions.y = pNode->Dimensions.y - ScalarY;
    pNewNode->Dimensions.z = pNode->Dimensions.z - ScalarZ;
    pNewNode->Position     = pNode->Position +
                             (pNode->Direction * pNode->Dimensions.z);
    pNewNode->Direction    = pNode->Direction;
    pNewNode->Right        = pNode->Right;
    pNewNode->Type         = bEnd ? BRANCH_END : BRANCH_SEGMENT;
    pNewNode->BranchSegment= pNode->BranchSegment + 1;
    pNewNode->Iteration    = (USHORT)Iteration;
    pNode->Child           = pNewNode;

    // Clamp to minimum size, this is an end of branch
    if ( pNewNode->Dimensions.x < 0.0f ) pNewNode->Dimensions.x = 0.0f;
    if ( pNewNode->Dimensions.y < 0.0f ) pNewNode->Dimensions.y = 0.0f;
    if ( pNewNode->Dimensions.z < 0.0f ) pNewNode->Dimensions.y = 0.0f;
```

We increment the passed BranchUID member before assigning it to the node so that this node has a unique ID. We then store the address of the parent node (the node passed into this function) in the new node's Parent pointer. We also subtract the three scalar values we calculated previously from the parent node's dimensions before assigning them as the new dimensions of the child node.

Notice how we assign the new node its position. Its position is the product of adding the parent node's direction vector scaled by the parent node's Z dimensions (the length of the parent node) to the position of the parent node. Essentially, we are generating the new position by placing the child at the position of the parent, then sliding the child node along the direction vector of the parent by the Z length. This is an important concept to grasp. The Z dimension of any node actually describes the length of the segment for which that node forms its base ring of vertices. Next (in the above code) we copy the direction vector and the right vector of the parent into the child; we will deviate these in a moment.

The value we assign to the node's Type member depends on whether the local bEnd boolean was set to true in our tests. If it was, then the new node becomes a BRANCH_END type and will be the last node in that branch. Otherwise, it is assigned the BRANCH_SEGMENT type which means it is just another normal segment added to the current branch. We also store in the node's BranchSegment member the value stored in the parent node increased by one. Remember, this value will start off at zero in every BRANCH_BEGIN node so that it contains the local index of the node within the branch. The iteration value passed into the function (which would have been incremented in a previous call) is also stored in the node and then the parent node's child pointer is assigned to point at this new node. Finally, we test to see if any of the node's dimensions are smaller than zero and clamp them to zero if this is the case. This could only potentially be the case for a node that was already determined to be a branch end node.

The new node which continues the current branch has now been populated and attached to its parent, adding another segment to the branch and another level to this portion of the hierarchy. We now decide whether the child node should be randomly deviated using our ChanceResult function and the TreeGrowthProperties::Segment_Deviation_Chance member. This property holds the probability that segments within the same branch will deviate. The code that performs the deviation is shown below.

```
    // Chance of changing direction
    if ( ChanceResult( m_Properties.Segment_Deviation_Chance ) )
    {
        // Deviate the node
        DeviateNode( pNewNode,
                     m_Properties.Segment_Deviation_Min_Cone,
                     m_Properties.Segment_Deviation_Max_Cone,
                     m_Properties.Segment_Deviation_Rotate);

        // Weight with parent
        pNewNode->Direction += pNode->Direction *
                               m_Properties.SegDev_Parent_Weight;

        // Weight with growth direction
        pNewNode->Direction += m_Properties.Growth_Dir *
                               m_Properties.SegDev_GrowthDir_Weight;

        // Normalize
        D3DXVec3Normalize( &pNewNode->Direction, &pNewNode->Direction );

    } // Change direction
```

If the ChanceResult function returns true, we enter the code block that deviates the child node's direction vector. We first call the DeviateNode method passing in the minimum cone angle, the maximum cone angle, and the rotation properties for node deviation within the same branch. When this

function returns, the new node's direction vector will have been randomly deviated from its parent. However, we also have other properties to configure to restrain that deviation. For example, next we calculate how much the parent node's direction vector should be factored into the deviated vector. We add the new node's direction vector and the parent node's vector multiplied by the SegDev_Parent_Weight property (usually a value between 0.0 and 1.0). The lower this weight value, the less influence the parent node's direction vector will have over the direction vector generated for the child. Next, we add the Growth_Dir vector to the new node's direction vector. Growth_Dir is a tree growth property describing a general growth direction for the tree. The influence it will have is determined by the SegDev_GrowthDir_Weight property. After we have added these two vectors to the deviated direction vector, we renormalize it. The vector is now deviated and influenced to some degree by the parent node's vector and the general growth direction.

At this point, our child node is complete. All we have to do is continue to process the branch to which it belongs by calling the BuildBranchNodes function again in a recursive call. This time, the new child node we have just created is passed as the parent node. The current value of the BranchUID variable is also passed so that it can continue to be incremented and assigned to nodes in this branch when they are generated in future recursions. It is at this point that we also increment the current Iteration value before passing it into the next recursion. With each call into the recursive process, the Iteration value will be incremented so that as we step down the hierarchy from branch node to child branch node, the iteration values of the nodes stored at that level in the hierarchy are increased.

```
    // Generate this new branch segment
    BuildBranchNodes( pNewNode, BranchUID, Iteration + 1 );

    // Reduce NewNode Count
    NewNodeCount--;
```

One thing to bear in mind is that this function is recursively called until the parent node passed in is a branch end node, at which point it exits. When the function returns to the current iteration, all the child nodes (and any child branches spawned from those child nodes) will have been generated.

Of course, our job is not yet done for this node. We may have determined earlier in the function that additional child branches should be spawned from the node we have just created. Notice in the above code that after we have added our new node, we decrement the local NewNodeCount variable to account for the fact that we have already processed one of the new nodes we needed to add.

If the NewNodeCount value is still non-zero after the previous decrement, it means that one or more child branches (branch start nodes) will be spawned from this node. Note that these will need to be added to the node's sibling list since the new branch nodes we are about to create and the child node we just added all share the same parent.

Before we add these new branch child nodes, we must determine whether the thickness of the current branch can accommodate child branches which may be protruding at extreme angles. When we create a new node to continue a branch, we shrink its size from its parent by one step (where a step is the value contained in ScalarX, ScalarY, and ScalarZ). However, when a root node of a new child branch is about to be placed, it is likely to be deviated by a much greater angle than the deviation performed on nodes within the same branch. Since the ring of vertices at the start node of a new branch will essentially be

placed *inside* the parent branch (i.e., positioned at the same position as the child node we just generated) we want to make sure that the child branch is thin enough such that when rotated by 45 degrees for example, the vertices placed at its first node do not pierce the skin of the parent branch (Figure 12.38).

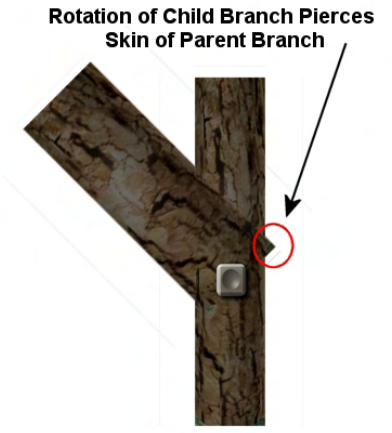


Figure 12.38

As you can see, our tree is a collection of disjoint meshes (one per branch). It is only the fact that we position the start nodes of those branches inside their parent branches that it looks like a single tree. However, as Figure 12.38 demonstrates, we need to make sure that any child branches we add are reduced enough in size with respect to the parent that we have room to rotate without the ring of vertices breaking the through the skin of the parent branch.

After some trial and error in the labs, we learned that by reducing the size of a new child branch by four steps (instead of one) we can generate children that can be freely rotated without much cause for concern. Therefore, instead of subtracting ScalarX, ScalarY, and ScalarZ from the size of the parent node as we did above when adding a node to continue the branch, when a new branch start node is spawned, we reduce its dimensions by four times that amount. So if we are at node N within a parent branch and a new branch is started at that node, the dimensions of that first node in the new branch will be equal to the thickness of node N+4 in the parent branch.

In the following code we assign this multiplier to a variable called fStepScale. We then perform tests to see if the dimensions of the parent node are smaller than the ScalarX, ScalarY, and ScalarZ values multiplied by the step scale. If the parent node is too thin to step down four levels in size, then we set the bEnd boolean to true.

```
    // Do we have enough room to split ?
    float fStepScale = 4.0f; // Reduce size by 4 times usual amount

    if ( pNode->Dimensions.x < (ScalarX * fStepScale) ) bEnd = true;
    if ( pNode->Dimensions.y < (ScalarY * fStepScale) ) bEnd = true;
    if ( pNode->Dimensions.z < (ScalarZ * fStepScale) ) bEnd = true;

    if ( bEnd ) NewNodeCount = 0;
```

If any of the dimensions of the parent node are so small that we would not be able to sufficiently reduce the size of the branch start nodes we are about to create, we set the NewNodeCount to zero. In this case, regardless of whether we determined a need to spawn child branches, no new branches will be started at this node.

If NewNodeCount is still larger than zero then we need to add one or more nodes to the parent node's child list in a loop. Each new node we are about to add in this step should be of the type BRANCH_BEGIN. For each node we allocate, we will assign it to the sibling pointer of the node that came before it in the loop, so we use the local pPrevNode pointer to store the address of the branch node that was allocated in the previous iteration of the loop. Initially we set the pPrevNode pointer to pNewNode, which is the node we allocated previously when adding another segment to the current branch. This node was assigned to the child pointer of the parent node, so any new nodes we add now should be attached to its sibling list.

```
    // Generate new split off branches ?
    BranchNode * pPrevNode = pNewNode;

    for ( i = 0; i < NewNodeCount; i++ )
    {

        BranchNode * pNewNode = new BranchNode;
        if ( !pNewNode ) continue;

        // Store node details
        pNewNode->UID          = BranchUID++;
        pNewNode->Parent       = pNode;
        pNewNode->Dimensions.x = pNode->Dimensions.x - (ScalarX * fStepScale);
        pNewNode->Dimensions.y = pNode->Dimensions.y - (ScalarY * fStepScale);
        pNewNode->Dimensions.z = pNode->Dimensions.z - (ScalarZ * fStepScale);
        pNewNode->Position     = pNode->Position +
                                 (pNode->Direction * pNode->Dimensions.z);
        pNewNode->Direction    = pNode->Direction;
        pNewNode->Right        = pNode->Right;
        pNewNode->BranchSegment= 0;
        pNewNode->Iteration    = (USHORT)Iteration;
        pNewNode->Type         = BRANCH_BEGIN;

        // Deviate the node
        DeviateNode( pNewNode,
                     m_Properties.Split_Deviation_Min_Cone,
                     m_Properties.Split_Deviation_Max_Cone,
                     m_Properties.Split_Deviation_Rotate );

        // Link the node
        pPrevNode->Sibling = pNewNode;

        // Generate this new branch
        BuildBranchNodes( pNewNode, BranchUID, Iteration + 1 );

        pPrevNode = pNewNode;

    } // Next New Branch Node
}
```

That was the final section of the BuildBranchNodes function. Notice that we assign each node a unique ID before incrementing the BranchUID variable. The dimensions we assign to each branch start node are the dimensions of the parent node reduced by four times the step reduction value used for inter-branch nodes. As before, the position of each node is simply calculated by sliding the new node from the parent node's position along the parent node's direction vector by the distance described in the parent node's Z dimension. The direction and right vectors are also copied straight from the parent node, and they will be deviated in a moment.

For each node we add in this loop we assign the BranchSegment member of that node to zero, because this member describes the index of the node relative to the start of the branch. Since we are adding branch start nodes here, the index of each new node we create will be zero. Of course, we also assign the type BRANCH_BEGIN to each node we create, as each one will begin a new branch mesh.

After we have assigned all the properties, we deviate the direction vector. We will usually want new branches to have a more obvious change in direction with respect to their parent branches so that we can see them sprouting out from the tree. When DeviateNode is called, we send in different minimum and maximum cone angles and polar rotation angles that we did before. These values specify the deviation range for nodes that start a new branch. Note that after we have deviated the direction vector of a new branch start node, we do *not* factor in the parent node's direction or the growth direction of the entire tree. The direction vector for this node will become the initial parent direction for the rest of the branch. The overall growth direction vector for the tree will be factored in eventually for the other nodes in the branch we are just starting. So even if the initial direction vector of the branch was (1,0,0), an overall growth direction of (0,1,0) would still make the other segments in the new branch start to grow upwards as they grow out.

Once the node has been created and populated, we attach it to the previous node's sibling pointer before the function calls itself recursively passing in the new node as the parent. When program flow returns back to the current instance of the function, the entire branch of nodes will have been created.

Before moving on, please make sure that you understand how the above function works by calling itself recursively to generate an entire tree of branches.

Phase One Complete

We have now covered all the code needed to implement phase one -- the building of the virtual tree. You will recall that phase one was initiated by CTreeActor::GenerateTree which was invoked by the application. The GenerateTree method called the GenerateBranches function to complete stage one. GenerateBranches called the recursive function BuildBranchNodes to build a virtual tree for each root branch node.

The GenerateTree function oversees the entire tree building process and we have only covered the first phase. Let us have another look at the GenerateTree method to remind ourselves of the functions it calls which we have yet to cover.

```
HRESULT CTreeActor::GenerateTree(   ULONG Options,
                                    LPDIRECT3DDEVICE9 pD3DDevice,
                                    const D3DXVECTOR3 & vecDimensions,
                                    const D3DXVECTOR3 & vecInitialDir ,
                                    ULONG BranchSeed /* = 0 */ )
{
    HRESULT            hRet;
    CAllocateHierarchy Allocator( this );

    // Validate parameters
    if ( !pD3DDevice ) return D3DERR_INVALIDCALL;

    // Release previous data.
    Release();

    // Store the D3D Device here
    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();

    // Store options
    m_nOptions = Options;

    // Generate the branches
    hRet = GenerateBranches( vecDimensions, vecInitialDir, BranchSeed );
    if (FAILED(hRet)) return hRet;

    // Build the frame hierarchy
    hRet = BuildFrameHierarchy( &Allocator );
    if (FAILED(hRet)) return hRet;

    // Build the bone matrix tables for all skinned meshes stored here
    if ( m_pFrameRoot )
    {
        hRet = BuildBoneMatrixPointers( m_pFrameRoot );
        if ( FAILED(hRet) ) return hRet;

    } // End if no hierarchy

    // All is well.
    return D3D_OK;
}
```

After we have the virtual tree from phase one, it is time to move onto the phase two. Phase two is the construction of the actor's skeleton and branch skins and is initiated with a call to BuildFrameHierarchy

Notice how BuildFrameHierarchy accepts one parameter -- a pointer to an ID3DXAllocateHierachy interface. The CAllocateHierarchy class we developed has not been changed since our previous discussions. It contains both the CreateFrame and CreateMeshContainer methods that we need to allocate frames for our actor's hierarchy and convert any regular meshes we create for the branches into API compliant skinned meshes. Let us now follow the path of execution into the BuildFrameHierarchy function.

BuildFrameHierarchy – CTreeActor

The BuildFrameHierarchy function is the gateway to the recursive process for phase two of the tree generation process. It has only one task -- to call the BuildNode function with parameters that initialize the recursive process.

```
HRESULT CTreeActor::BuildFrameHierarchy( ID3DXAllocateHierarchy * pAllocate )
{
    D3DXMATRIX mtxRoot;

    // Initial frame is identity
    D3DXMatrixIdentity( &mtxRoot );

    // Build the nodes
    return BuildNode( m_pHeadNode, NULL, NULL, mtxRoot, pAllocate ) ;
}
```

The BuildNode function is passed the root node of the branch node hierarchy constructed in phase one. This function also needs to be passed a combined frame matrix that is initially set to identity (as the root node has no parent and we have not generated any frames yet). We also pass in the CAllocateHierarchy interface pointer so that the BuildNode function will be able to use its methods to generate meshes and frames. When this function returns, the entire frame hierarchy will have been constructed and populated with skinned meshes (one per branch).

BuildNode – CTreeActor

The BuildNode function is a function that recurses until the frame hierarchy and all its meshes have been created. Before we look at the code, let us briefly examine the design specifications so that we know what this function must do.

Frame Hierarchy Creation

This function will need to recursively step through the branch node hierarchy (the virtual tree) and examine each node. If a BRANCH_BEGIN node is encountered then we know that this node will be the root bone of a given mesh. Also, if we reach any type of node (except BRANCH_END) that has an iteration where Node->BranchSegment % Bone Resolution = 0, we know that we have skipped the correct number of nodes and it is time to add another bone. Remember, the bone resolution property tells this procedure how frequently it should convert branch nodes into actual bones. The BranchSegment member of a node contains its zero based index relative to the start of the branch.

Once the function determines that the current branch node is one that should be a bone, we have to allocate a new D3DXFRAME and attach it to the actor's frame hierarchy. Obviously, in the very first iteration the actor will have no hierarchy, so the first frame we create will be the root frame of the actor (the actor's m_pRootFrame member will point at this frame).

Once we have created a frame for a given node, we will also set that branch node's BoneNode boolean to true so that we will know later on that this node is a node that created a bone. This will be needed when building the skin and mesh data. We will also store a pointer to the frame we have just created in the branch node's pBone pointer. Once again, this is so we can access it later and pair branch nodes with frames.

Now that a frame has been allocated for this branch node, we need to populate it with information. Obviously, we wish this bone to be stored in the exact same position as the branch node it was created from. However, we cannot just copy over the position and orientation from the branch node into the frame's matrix. The branch node stores the position and orientation of the node as an absolute transformation from the origin of tree space, but we know that the frames must have their matrices defined in parent relative space. To address this, we will pass through the recursive process a combined frame matrix and the process will work as follows…

If we are currently adding a fifth frame to the actor's hierarchy, we will have a matrix that contains the combined transformations of the first four. We will also create a matrix for the current frame that contains the absolute position and orientation information copied over from the branch node. We now have matrix A (a combined matrix of frames 1 to 4) and matrix B (the absolute position of the current frame as taken from the branch node). We can calculate the relative matrix by inverting matrix A and multiplying with matrix B.

Relative Matrix = $B\ A^{-1}$

Essentially, we are subtracting the absolute position of the 4^{th} frame (the parent) from the absolute position of the 5^{th} frame (copied from the branch node) which leaves us with a matrix describing the transformation of the fifth frame relative to the fourth. Now you know how to construct relative matrices. Once we have this relative matrix, we will store it in the matrix of the frame.

Of course, we will need to pass the current combined matrix up to the point of the current branch node down to the child nodes. Therefore, using the above example, once we have calculated the relative matrix for frame 5, we will multiply that matrix with the combined matrix of frames 1-4 (passed into the function) so that we now have a combined matrix of transformations for frames 1 through 5. We will then pass that matrix into the next recursion so that it can be 'subtracted' from the absolute position of the 6^{th} frame to generate the relative matrix with respect to its parent, frame 5. Because not all branch nodes will become bones, we need to make sure that even when we do not add a frame to the hierarchy for a node that we are processing, we still pass the combined matrix into the next recursion.

For example, imagine we have a bone resolution of 5. At the first frame, we hit a BRANCH_BEGIN node, so we add a frame (the first frame) to the actor's hierarchy. Given the bone resolution, we will not be adding another frame for this branch until we traverse another five branch nodes. However, when we hit that next branch node where a frame must be added, we will need access to the combined matrix that was generated at the last branch node a frame was created for. We need it so that we can calculate the relative matrix for the new frame. Therefore, each time the function is called, we will pass in a combined frame matrix containing

the concatenation of all the frames we have added to the hierarchy up to that point. When the branch node we are processing is one that we will create another frame in the hierarchy for, that combined matrix will be updated by combining it with the relative matrix generated for the new frame. This updated combined matrix can then be passed down to the children and used further down the branch when another frame is added. If the branch node we are currently processing is a non-bone branch node, we will simply pass along the combined matrix un-modified to the children.

Mesh Creation

The function will do more than just create frames for the actor's frame hierarchy; it will also build a CTriMesh for each branch. Once again, the action taken by the function is slightly different depending on whether we are processing a BRANCH_START node or a normal branch node.

1. If the branch node is of type BRANCH_BEGIN, we have to create a new empty mesh. We then add the initial ring of vertices to this mesh before passing this mesh down to the child branch nodes.

2. If the branch node is not of type BRANCH_BEGIN then we are currently stepping down a branch for which a mesh has already been created. All we have to do is add another ring of vertices to that mesh at the position represented by the branch node.

Every time the function calls itself for a non-BRANCH_BEGIN node, it will be passed a pointer to the mesh that was allocated when the BRANCH_BEGIN node was encountered. Therefore, we use a similar transport mechanism for the mesh as we do for the combined matrix. The branch begin node allocated the mesh, and from that point on this mesh is passed down to each child node where rings of vertices are added. By the time the BRANCH_END node has been created, we will have added a ring of vertices to the mesh for every node in that branch.

Of course, when stepping through branch child nodes and building the mesh, we may find that a node exists in a child's sibling list. This tells us that one or more BRANCH_BEGIN nodes (new child branches) start at this node. For each of these BRANCH_BEGIN nodes, we must do the same thing as before -- create a new mesh and then recursively pass it down to each of its children so that they can add their vertices. So you can imagine that as the recursive process is underway, there can be many meshes allocated but only partially built.

For example, imagine a simple case where we have a root branch with six nodes. At the first node (BRANCH_BEGIN) we create a new CTriMesh and pass it down to each of the five children in the list. By the time we have reached the branch end node, the branch will have had five rings of vertices added and one tip vertex. However, let us also assume that this root branch has a child branch that starts at node 3 (another BRANCH_BEGIN node in a sibling list with node 3 of the root branch). We would travel into that branch and process its BRANCH_BEGIN node. This would mean allocating a new mesh for this second branch and stepping through its children adding their vertex rings.

At some point we will reach the branch end node of this second branch and all vertices will have been added. The initial mesh for the root branch is still only partially complete because we took a detour after processing node 3. When the recursive process returns, we find ourselves back in the instance of the function that was originally processing node 3 of the root branch. We would now continue down to the remaining branch nodes in the root branch and complete our adding of vertices to the root branch mesh.

Figure 12.39 shows the flow of the recursive process when walking the branch node hierarchy and building and populating branch meshes.

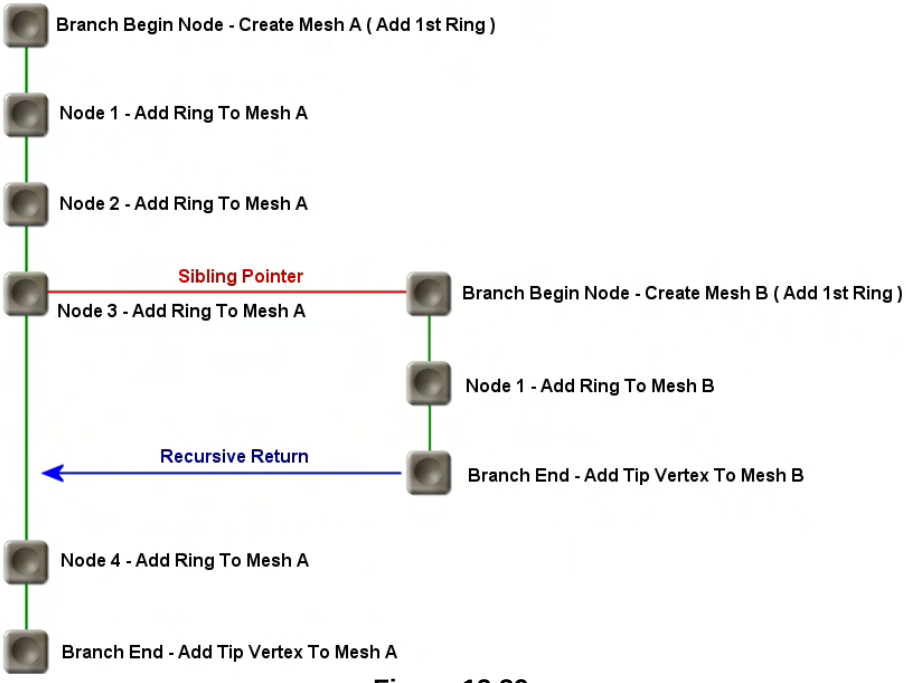


Figure 12.39

Regardless of how many branches your tree has and how many meshes it ultimately creates, the root mesh (the mesh of the trunk) will always be the first mesh allocated and the last mesh completed. We can see in Figure 12.39 that Mesh A was allocated first, but Mesh B was completed first. If you imagine more levels of recursion (child branches coming off of branch B), then you should be able to see that the last meshes to be allocated and the first to be completed are always the meshes for the smaller branches at the deepest levels of the tree.

When we add vertices to a mesh at a given node we will also add the indices to the mesh index buffer to create the branch segment (a cylinder of triangles) between the ring of vertices added at the current node and the ring of vertices added at the previous node. Remember, after the initial branch begin node, every node thereafter inserts vertices that create a new branch segment. Each ring of vertices inserted at a node (excluding the BRANCH_BEGIN node) forms the top row of vertices of the current segment being added and the bottom row of vertices for the next segment added when the child node is processed.

At the end of the BuildNode function we will once again test to see if the node we are currently processing is a BRANCH_BEGIN node. We know that if it is, then this instance of the function will have already recursively visited the child node, which would have visited its child node, and so on. Suffice to say, at this point, the recursive process has returned back to this instance of the function and the entire child branch will have had its vertices and indices added.

Recall that when we add vertices and indices to a CTriMesh, it stores them temporarily in its internal system memory arrays. Once we are sure that we have added all the vertices and index data, we call CTriMesh::BuildMesh to copy these arrays into vertex and index buffers and create the underlying D3DX mesh. Once done, we can pass the ID3DXMesh into the CAllocateHierarchy::CreateMeshContainer function to convert it into a proper skinned mesh before attaching it to the BRANCH_BEGIN frame in the hierarchy.

Of course, we must send other parameters into CreateMeshContainer, such as the material and texture lists the mesh uses. This list has only one element -- the material and the texture filename set by the application in the call to CTreeActor::SetBranchMaterial. CreateMeshContainer will take care of executing the relevant callbacks to load and store this texture and material combination.

When we covered the CAllocateHierarchy::CreateMeshContainer function, we also learned that if an intended skinned mesh is being passed in, we will also be supplied with a pointer to an ID3DXSkinInfo interface. It is this interface that provides the information needed to create the skin, such as which vertices are influenced by which bones and by what weight. It also contains the name of each bone and its accompanying bone offset matrix. For our tree, we will have to calculate all of this data ourselves, store it in an ID3DXSkinInfo object, and pass it to CreateMeshContainer. Therefore, before we call the CAllocateHierarchy::CreateMeshContainer method, we will issue a call to a helper function called CTreeActor::BuildSkinInfo to create this object for us. That allows us to treat this function like a black box until we have finished discussing the BuildNode function. We will then take a look at the BuildSkinInfo method and see exactly how we build the mapping information between vertices and bones and how we calculate the bone offset matrices for each frame used as a bone by the branch.

We are now ready to cover the CTreeActor::BuildNode function. First, let us have a look at its parameter list and describe what each parameter will be used for. This will make it much easier to then explain the code itself. The function accepts five parameters, which are listed below.

BranchNode *pNode
This is a pointer to the branch node that the instance of the BuildNode function will process. When the BuildNode function is initially called from the BuildFrameHierarchy function, a pointer to the root branch node of the virtual tree hierarchy will be passed. The function will recursively call itself using this parameter to step down to child nodes and along to sibling nodes.

It is this node that will contain the position and orientation of the ring of vertices that needs to be placed in this iteration of the function. If the node is of type BRANCH_BEGIN, the position and orientation of this node will be used to add a new frame to the actor’s frame hierarchy.

D3DXFRAME * Parent
This parameter will be NULL in the initial call. For all other recursive steps it will contain a pointer to the previous frame that was generated for the current branch (i.e., the frame that was last added to the hierarchy) and thus serves as the parent to the next frame being built along the branch. Not all nodes will create frames, so we must pass this pointer down to child nodes even if no frame was created for the current node.

CTriMesh * pMesh
This member will be set to NULL when the function is initially called from the BuildFrameHierarchy function. For all other iterations it will contain a pointer to the mesh that has been allocated for the current branch being constructed.

D3DXMATRIX &mtxCombined
This parameter will be set to an identity matrix when the function is initially called from the BuildFrameHierarchy function. For all other instances of the function, it will contain the combined (absolute) matrix of all the frames we have created so far along that path of the frame hierarchy.

When the function is processing a node that does not result in the generation of a new frame, this pointer should simply be passed unaltered into the child nodes. This allows us to continually pass the current combined matrix down through the branch node until it reaches a node where a new frame is to be generated. It is then used as part of the process to calculate that frame's relative matrix as discussed earlier in the chapter.

When the function recursively calls itself to process a sibling, it should also simply pass along the same combined matrix to the sibling that was passed into the function. The sibling frames will all share the same parent frame and therefore need to use the same parent absolute matrix to calculate their relative matrices.

ID3DXAllocateHierarchy * pAllocate
As we saw when we looked at the BuildFrameHierarchy function, this parameter always contains a pointer to an instance of our CAllocateHierarchy class. We use this object to call its CreateFrame and CreateMeshContainer callback methods whenever we wish to create a new frame in the hierarchy or build a (skinned) mesh.

With the parameter list now behind us, let us now look at the code a section at a time.

The first section of the code tests to see if the current node being processed is a node that should generate a new frame in the actor's hierarchy. The test for this is quite simple: if the node is of type BRANCH_BEGIN then this node represents the start of a new branch mesh and the initial bone of the branch. In this case, we definitely want to add a bone to the actor's hierarchy at this location. The other case in which we decide to create a new bone is if the modulus of the branch node's BranchSegment member with the tree's Bone_Resolution property is zero. Bear in mind that the BranchSegment member of a node contains the zero-based local index of the node within its branch. If a branch has 10 nodes, the BranchSegment member for each node would be in the range of 0-9, respectively. If the bone resolution was 3 for example, we would only add bones at the following nodes:

Node 1		BRANCH_BEGIN			(Bone Created)
Node 2	=	BranchSegment	Mod	Bone_Resolution	
	=	1	Mod	3 = 1	
Node 3	=	BranchSegment	Mod	Bone_Resolution	
	=	2	Mod	3 = 2	
Node 4	=	BranchSegment	Mod	Bone_Resolution	(Bone Created)
	=	3	Mod	3 = 0	
Node 5	=	BranchSegment	Mod	Bone_Resolution	
	=	4	Mod	3 = 1	
Node 6	=	BranchSegment	Mod	Bone_Resolution	
	=	5	Mod	3 = 2	
Node 7	=	BranchSegment	Mod	Bone_Resolution	(Bone Created)
	=	6	Mod	3 = 0	
Node 78	=	BranchSegment	Mod	Bone_Resolution	
	=	7	Mod	3 = 1	
Node 9	=	BranchSegment	Mod	Bone_Resolution	
	=	8	Mod	3 = 2	
Node 10	=	BranchSegment	Mod	Bone_Resolution	(Branch End = No Bone)
	=	9	Mod	3 = 0	

Remember that modulus operation returns the remainder of A divided by B as an integer. So as you can see, only when the modulus returns zero (evenly divisible) have we hit the bone resolution boundary where a new bone should be added. Of course, we always create a bone at the start of a branch. Notice that when we add the 10^{th} node, the modulus does indeed return zero, but we do not add a bone here. That is because it is the end of the branch and a bone would be unnecessary.

Let us have a look at the first section of the function and then we will discuss it.

```
HRESULT CTreeActor::BuildNode( BranchNode * pNode,
                               D3DXFRAME * pParent,
                               CTriMesh * pMesh,
                               const D3DXMATRIX & mtxCombined,
                               ID3DXAllocateHierarchy * pAllocate )
{
    HRESULT     hRet;
    CTriMesh   *pNewMesh  = NULL, *pChildMesh  = NULL;
    LPD3DXFRAME pNewFrame = NULL,  pChildFrame = NULL;
    D3DXMATRIX  mtxBranch, mtxInverse, mtxChild;
    D3DXVECTOR3 vecX, vecY, vecZ;
```

```
    TCHAR       strName[1024];

    // What type of node is this

     bool bIgnoreNodeForBone = (pNode->Type == BRANCH_END);
    if ( pNode->Type == BRANCH_BEGIN ||
        ((pNode->BranchSegment % m_Properties.Bone_Resolution) == 0 &&
         !bIgnoreNodeForBone) )
        {
        // We get here if either this is a new branch node, OR the
        // bone resolution is such that a new bone based frame is being
        // created here. Note: We don't drop here if this is an END node
        // since creating a new bone frame here would be pointless.

        // Generate frame name
        _stprintf( strName, _T("Branch_%i"), pNode->UID );

        // Allocate a new frame
        hRet = pAllocate->CreateFrame( strName, &pNewFrame );
        if ( FAILED(hRet) ) return hRet;

        // If there is a parent, store the new frame as a child.
        if ( pParent )
        {
            // Attach to head of parent's linked list
            pNewFrame->pFrameSibling  = pParent->pFrameFirstChild;
            pParent->pFrameFirstChild = pNewFrame;

        } // End if has a parent
        else
        {
            // Store at the actor's root
            pNewFrame->pFrameSibling = m_pFrameRoot;
            m_pFrameRoot             = pNewFrame;

        } // End if no parent found
```

In the above code we first determine whether a bone should be created for the current node using the techniques previously discussed. If it is determined that we should, we first build a name for that frame. Remember that in order for our frames to be animated, they must have names that will eventually match the animations inside an animation set. We need each frame in the hierarchy to have a name that is unique from any other frame, which was why we took the trouble to assign each branch node a unique numerical ID during the building of the branch node hierarchy. Thus, we build a name for the frame in the format Branch_*n* , where *n* is the ID of the branch node. We use the _stprintf (C standard runtime library) function to format the string and store the result in the local char array strName.

Now that we have the name for the frame we are about to allocate, we pass it into the CAllocateHierarchy::CreateFrame function. As we know, this is a method of our CAllocateHierarchy class that simply allocates a new D3DXFRAME derived structure, safely initializes its members and returns it. The new frame is returned to us in the pNewFrame local pointer.

Once we have our new frame, the code determines whether a parent frame currently exists (it will for all iterations of the function other than the first). If the pParent parameter is not NULL, then it contains a

pointer to the previous frame that was generated during the building process. When this is the case, we add our newly created frame to its child list. Of course, the parent frame may already have children (other branch start frames for example) so we are careful to simply add our new node to the head of the child list thus keeping the child list intact. In this case, we assign the new frame's sibling pointer to point at the current child pointer of the parent (which may point to a list of siblings), and assign the parent's child pointer to point at our new frame. In doing so, we have just added our new frame to the head of the list of child frames for the parent.

If the pParent pointer is NULL, then this is our first time through the function and the frame we have just allocated is the root frame. In this case we assign the actor's m_pFrameRoot pointer to point to our new frame, which is now the root frame for the actor's entire hierarchy. Notice however that we are still careful even in this case to attach the current value of m_pFrameRoot to the new frame's sibling pointer. Remember, there may be multiple root frames in our hierarchy that exist as siblings at the root level. Just because this frame has no parent does not mean that there is nothing useful being pointed to by the m_pFrameRoot pointer -- it may be the sibling root frame of another trunk branch -- so we should keep this list intact also.

Having created the new frame, we will also set the current branch node's BoneNode boolean to true so that we will know later that this branch node has had a bone created from it. We will also assign the branch node's pBone pointer to point at our new frame so that we have access later on.

```
        // For skin info building notify that this is the start of a new bone
        pNode->BoneNode = true;

        // Store which bone we're assigned to in the node
        pNode->pBone = pNewFrame;

        // Update the frame we will pass to the child with our new frame
        pChildFrame = pNewFrame;
```

Notice at the bottom of the above code, we assign a local pointer (pChildFrame) to the new frame we have just allocated. Because we have created a new frame, we know that it will be the parent frame of the next frame that is created further down the branch. So this pointer will serve as the frame that is passed to the child node as its parent.

We have now created a frame and attached it to the hierarchy but it currently requires initialization. Its matrix is just an identity matrix, so we need to compute the parent relative matrix. We do not know this information at the moment but we do have the absolute position of the current branch node the frame was created from, as well as the node's direction and right vectors. These vectors describe the alignment of the branch node's local Z and X axes with respect to tree space, and if we perform the cross product between the branch node's right and direction vectors we will generate the third axis of the branch node's local coordinate system.

```
        // Store / generate the vectors used to build the branch matrix
        vecX = pNode->Right;
        vecZ = pNode->Direction;
        D3DXVec3Cross( &vecY, &vecZ, &vecX );
```

At this point, we now know the orientation of the branch node's local axes as well as its position in tree space. This is all we need to build a tree space (absolute) matrix for the current frame.

```
        // Generate the frame matrix for this branch
        D3DXMatrixIdentity( &mtxBranch );
        mtxBranch._11 = vecX.x; mtxBranch._12 = vecX.y; mtxBranch._13 = vecX.z;
        mtxBranch._21 = vecY.x; mtxBranch._22 = vecY.y; mtxBranch._23 = vecY.z;
        mtxBranch._31 = vecZ.x; mtxBranch._32 = vecZ.y; mtxBranch._33 = vecZ.z;
        mtxBranch._41 = pNode->Position.x;
        mtxBranch._42 = pNode->Position.y;
        mtxBranch._43 = pNode->Position.z;
```

By placing the three axis vectors and the position into a matrix (using the specified format) we have created a single tree space matrix that describes the position and orientation of the branch node. In this coordinate system the tree root would be at the origin, but hopefully you can see that this matrix is no different from any other world matrix we would generate. Essentially, taking a point at the origin of the tree space coordinate system (0,0,0) and multiplying it with this matrix would transform the point to the position of the current node.

So, we have a matrix that describes the absolute position of our new frame, but we want it to be a parent relative matrix. We also have the combined matrix passed into the function (mtxCombined) which describes the absolute matrix of the parent frame. As discussed earlier, if we take the inverse of this combined matrix and multiply it with our current matrix, we will *essentially* subtract from our matrix all the combined transformations for all frames up to and including the parent frame. This will leave our matrix storing only the transformation 'difference' between the parent frame and the current frame's position/orientation. In other words, we have a matrix that describes our new frame as position and rotation offsets from the parent frame. In short, we have a parent relative matrix.

> **Note:** For a more technical description of what is happening here, you can refer back to Module I where we discussed the relationship between matrices, inverse matrices, and moving into and out of local coordinate systems.

```
        mtxChild = mtxBranch;

        D3DXMatrixInverse( &mtxInverse, NULL, &mtxCombined );
        D3DXMatrixMultiply( &mtxBranch, &mtxBranch, &mtxInverse );

        // Store the parent relative matrix in the frame
        pNewFrame->TransformationMatrix = mtxBranch;
```

As you can see in the above code we invert the current combined parent frame matrix and multiply it with our current absolute matrix (mtxBranch), storing the parent relative result back into mtxBranch. This is the matrix we need, so we assign our new frame's TransformationMatrix pointer to point at it. Our new frame has now been attached to the hierarchy and has received the correct matrix.

Note that the first step we took was making a copy of the absolute frame matrix (mtxBranch) to a local variable (mtxChild) before we modify it to become relative. Why? Well, think about what the mtxCombined matrix that was passed into this function contains -- the absolute matrix of the parent. When we recur down into the child node of the current node and generate a child frame, we will need

the combined matrix passed into that function to be the absolute matrix of the current frame we have just generated. Therefore, mtxChild will contain the combined matrix of the parent when processing the child node.

At this point we have created the new frame, attached it to the hierarchy, and assigned it a parent relative transformation matrix. We have also stored a local copy of the absolute transformation for our new frame so that it can be passed down to a child as the mtxCombined parameter and used to generate its relative matrix. The pChildFrame local frame pointer stores the address of our newly created frame so that it can be passed into the children of this branch as the pParent parameter. So between the mtxChild and pChildFrame local variables, we are ready for the next level of recursion into the child node.

> **Note**: At this point, please remember that we are still inside the code block that is executed only if a new frame is to be generated at this node.

In the next section of code we need to determine two things. If the current node we have just created a frame for is of type BRANCH_START then we know that this represents the start of a new branch mesh. We also need to determine what mesh we are going to pass down to the child node in the recursion. As with the mtxChild and pChildFrame local variables, we use a local variable called pChildMesh to store the mesh we need to pass down the branch.

If we are at a branch start node, we will ignore the mesh parameter passed into the function and create a new CTriMesh. We assign the pChildMesh local pointer to point at this new mesh. This will be the mesh we pass in as the mesh parameter when BuildNode calls itself for the child nodes in this new branch. If this is not a branch start node, then no new mesh needs to be created; it is just another node in a branch mesh that is already under construction. In this case, the pMesh parameter passed into the function will contain the address of the mesh that this node should pass down to its child, and the mesh it should add vertices to.

```
        // We only start creating the 'skin' mesh if this is a beginning branch
        if ( pNode->Type == BRANCH_BEGIN )
        {
            // We're going to start building a mesh, so create a new one
            pNewMesh = new CTriMesh;
            if ( !pNewMesh ) return E_OUTOFMEMORY;

            // Setup the new mesh's vertex format
            pNewMesh->SetDataFormat( VERTEX_FVF, sizeof(USHORT) );

            // Update the mesh we will pass to the child with our new mesh
            pChildMesh = pNewMesh;

        } // End if BRANCH_BEGIN
        else
        {
            // Reuse the mesh we were passed.
            pChildMesh = pMesh;

        } // End if other branch node type
```

At this point, whether we have allocated a brand new mesh, or are simply continuing the process of adding vertices to the mesh passed in for a current branch, the pChildMesh pointer will point to the mesh into which we will add the ring of vertices for this node.

Luckily for us in the short term, we use the CTreeActor::AddBranchSegment function to accomplish this task. It is passed the current node and the mesh we wish to add another ring of vertices to. We will discuss the code for the AddBranchSegment function next. Suffice to say, when it returns, a new branch segment will have been added to the branch mesh.

```
        // Add the ring for this segment in this frame's combined space
        hRet = AddBranchSegment( pNode, pChildMesh );
        if ( FAILED(hRet) ) { if (pNewMesh) delete pNewMesh; return hRet; }

    } // End if adding new frame
```

The curly brace at the bottom of the above code closes the code block that we have been examining so far. This is the code block that is only executed if the current branch node we are processing is a node for which a bone must be created and added to the hierarchy.

The next section of code is the else block of that conditional. It is executed if we are processing a node that will not add another bone to the frame hierarchy. However, we must still add a ring of vertices since every node (not just bone nodes) represents part of the branch.

```
    else
    {
        // Store which bone we're assigned to in the node
        pNode->pBone = pParent;

        // Add the ring for this segment in this frame's combined space
        hRet = AddBranchSegment( pNode, pMesh );
        if ( FAILED(hRet) ) { return hRet; }

        // Since no new frame is generated here, the child will receive
        // the same frame, mesh and matrices that we were passed.
        pChildFrame = pParent;
        pChildMesh  = pMesh;
        mtxChild    = mtxCombined;

    } // End if continuing build of previous mesh
```

As you can see, we call the AddBranchSegment function to add a ring of vertices to the mesh for this node. Notice that we pass in the pMesh parameter which contains the pointer to the mesh that has been passed into this function by the node's parent. Also notice how we set the local variables that will become the parameters for the child recursion. pChildFrame is set to the pParent frame pointer passed in since no new frame has been created and the parent frame should be passed unmodified into the child. The only time we ever wish to change the parent frame is when a new frame is inserted into the hierarchy. Finally, we set the mtxChild matrix to absolute parent matrix (mtxCombined) passed into the function. Since the parent frame is unchanged, so too is its absolute transformation. Thus, all we are doing is taking the input mesh, parent frame, and parent matrix, and preparing them to be passed down to the child.

At this point, we will now process the sibling list if a sibling exists at the current node.

```
    // Build the nodes for child & sibling
    if ( pNode->Sibling )
    {
        hRet = BuildNode( pNode->Sibling, pParent, pMesh,
                          mtxCombined, pAllocate );

        if (FAILED(hRet)) { if ( pNewMesh ) delete pNewMesh; return hRet; }

    } // End if has sibling
```

The BuildNode function calls itself recursively for the sibling node. It passes in the same parent and parent matrix as the current node (all siblings share the same parent) as well as the mesh pointer that was passed into the function.

Because of the way we organized our branch node hierarchy, the node of a branch will always be at the head of a sibling list. All other branch nodes in that sibling list (if any exist) will begin new branches. Therefore, strictly speaking, the mesh we are passing into the sibling will never be used, because as soon as we step into a sibling it will be of type BRANCH_START. This means that it will allocate its own CTriMesh for that branch and pass it down to each of its own children. However, just to safeguard against the fact that the application may have changed the order of the sibling list, we pass it the mesh. If a non-BRANCH_START node were it to exist somewhere in the sibling list, our code would still handle it correctly. The important thing to realize at this point is that when the above function call returns, all child branches that start at this node will have been *fully* built. Each sibling branch will have been traversed and created and so too will have any child branches of those child branches, and so on.

We still need to step though the remaining nodes of the current branch, so we issue the recursive call to the child pointer.

```
    if ( pNode->Child )
    {
        hRet = BuildNode( pNode->Child, pChildFrame, pChildMesh,
                          mtxChild, pAllocate );

        if (FAILED(hRet)) { if ( pNewMesh ) delete pNewMesh; return hRet; }

    } // End if has child
```

Notice this time that when we step into the child node, we pass in the pChildFrame, pChildMesh, and mtxChild local variables. What these variables store depends on whether a new frame and mesh was created at this node. Below we clarify what these variables will hold depending on the status of the current node:

Current Node = BRANCH_BEGIN node
pChildFrame = New frame added in this function
pChildMesh = New Mesh allocated in this function
mtxChild = Absolute matrix of the frame added in this function

Current Node = Normal Node (New Bone Node)
pChildFrame = New frame added in this function
pChildMesh = pMesh passed into this function from parent (pMesh)
mtxChild = Absolute matrix of the frame added in this function

Current Node = Normal Node (Non Bone Node)
pChildFrame = Parent frame passed into this function (pParent)
pChildMesh = pMesh passed into this function from parent (pMesh)
mtxChild = Absolute matrix of the frame added in this function (mtxCombined)

When the call to the child node returns, every remaining child node in the current branch will have been visited and will have added their rings of vertices to the mesh.

If the current node is not a BRANCH_BEGIN node then we have nothing left to do in this function. However, if the current node was a BRANCH_BEGIN node, then the CTriMesh for this branch will have had all the required vertices and indices added to it. It is then time to build the underlying ID3DXMesh and pass it to CAllocateHierarchy::CreateMeshContainer to have it converted to a skinned mesh. We will attach the returned mesh container to the branch start frame in the hierarchy.

```
    // If this was a 'begin' node, we can now build the mesh container
    // since all of the mesh segments should have been created by the
    // recursive calls.
    if ( pNode->Type == BRANCH_BEGIN )
    {
        D3DXMATERIAL        Material;
        D3DXMESHDATA        MeshData;
        D3DXMESHCONTAINER * pNewContainer       = NULL;
        DWORD             * pAdjacency          = NULL;
        LPD3DXBUFFER        pAdjacencyBuffer    = NULL;
        LPD3DXSKININFO      pSkinInfo           = NULL;

        // Generate mesh containers name
        _stprintf( strName, _T("Mesh_%i"), pNode->UID );

        // Generate the skin info for this branch
        hRet = BuildSkinInfo( pNode, pNewMesh &pSkinInfo );

        if ( FAILED(hRet) ) { delete pNewMesh; return hRet; }

        // Signal that CTriMesh should now build the mesh in software.
        pNewMesh->BuildMesh( D3DXMESH_MANAGED, m_pD3DDevice );
```

The first thing we do is call the CTreeActor::BuildSkinInfo function. We will look at this function shortly, but for now just know that it will create a new ID3DXSkinInfo object and will fill it with the vertex/bone mapping data. When the function returns, the pSkinInfo local variable will contain a pointer to the interface of this object. After building the skin info, we call CTriMesh::BuildMesh so that the vertices and indices we have been adding to the mesh (via the AddBranchSegment function) are used to construct the ID3DXMesh.

With our regular mesh now complete, we need to send it into the CreateMeshContainer function wrapped inside a D3DXMESHDATA structure. We set the Type member of the D3DXMESHDATA member to D3DXMESHTYPE_MESH so that the mesh container knows it is being given a regular ID3DXMesh.

```
        // Build the mesh data structure
        ZeroMemory( &MeshData, sizeof(D3DXMESHDATA) );
        MeshData.Type  = D3DXMESHTYPE_MESH;

        // Store a reference to our build mesh.
        // Note: This will call AddRef on the mesh itself.
        MeshData.pMesh = pNewMesh->GetMesh();

        // Build material data for this tree
        Material.pTextureFilename = m_strTexture;
        Material.MatD3D           = m_Material;
```

The CreateMeshContainer function also expects to be passed the texture filename and material for each subset. For our tree mesh, we have only one subset so we set up a single D3DXMATERIAL structure to store the texture filename and the material that the application set for the CTreeActor (via SetBranchMaterial).

Finally, before calling CreateMeshContainer we calculate the adjacency information for the mesh we are about to pass. We use the CTriMesh::GenerateAdjacency function, which generates the adjacency information and stores it internally inside an ID3DXBuffer object. We then call CTriMesh::GetAdjacencyBuffer which returns a pointer to its ID3DXBuffer interface and follow that with a call to ID3DXBuffer::GetBufferPointer to get a pointer to the actual adjacency information. We pass all this information into the CreateMeshContainer function which then creates a new skinned mesh (using the appropriate supported skinning method) and returns the results in the pNewContainer pointer passed in as the final parameter.

```
        // Retrieve adjacency information
        pNewMesh->GenerateAdjacency( );
        pAdjacencyBuffer = pNewMesh->GetAdjacencyBuffer();
        pAdjacency       = (DWORD*)pAdjacencyBuffer->GetBufferPointer();

        // Create the new mesh container
        hRet = pAllocate->CreateMeshContainer( strName,
                                               &MeshData,
                                               &Material,
                                               NULL,
                                               1,
                                               pAdjacency,
                                               pSkinInfo,
                                               &pNewContainer );

        // Release adjacency buffer
        pAdjacencyBuffer->Release();

        // Release the mesh we referenced
        MeshData.pMesh->Release();
```

```
        // Release the skin info
        if (pSkinInfo) pSkinInfo->Release();

        // Destroy our temporary child mesh
        delete pNewMesh;

        // If the mesh container creation failed, bail!
        if ( FAILED(hRet) ) return hRet;

        // Store the new mesh container in the frame
        pNewFrame->pMeshContainer = pNewContainer;

    } // End if beginning of branch

    // Success!!
    return D3D_OK;

}
```

When CreateMeshContainer returns, this branch will have been created as a skinned mesh and attached to a mesh container. We can then delete the original CTriMesh and the adjacency buffer before assigning the BRANCH_BEGIN frame's pMeshContainer pointer to our newly generated mesh container (which stores our branch skin).

While this was quite a complex function, remember that it is responsible for the entire hierarchy and mesh building process. When the initial instance of the function (called from BuildFrameHierarchy) returns the actor's hierarchy will be complete.

It is now time to look at the helper functions that were called from BuildNode to assist in accomplishing its required tasks. The first function we will look at is the AddBranchSegment method.

AddBranchSegment – CTreeActor

This method is passed a branch node and a pointer to a CTriMesh and it has two tasks it must perform. It must add the ring of vertices to the mesh that this node represents and it must add the indices to the mesh which connect vertices of the parent node to the vertices of the current node. This is how we form our cylinder (a branch segment) faces between the two nodes.

Things are not quite a simple as they first seem because the position of the node and the orientation of its vectors are defined in tree space. All node positions are relative to the root node of the virtual tree and therefore we might say that the node's positions and orientations are specified absolutely throughout the tree. The problem we must overcome is that each branch will be a separate mesh and we are going to want our vertex positions defined in the mesh's model space.

In the case of a branch mesh, we should consider the root node of that branch to be the origin of model space. Therefore, we want the vertex positions we add to this mesh to be defined relative to the root node of the branch, not to the root node of the entire tree. Essentially, we need to drag the root node of a

branch back to the origin of the coordinate system (dragging all its child nodes with it) so that the start node of the branch is at (0,0,0) and its local axes are aligned with the X,Y, and Z axes of the coordinate system. If we can do that, then we will have the positions and orientations of all its child nodes (in the same branch) defined in mesh/model relative space. In model space then, the start node of a branch is at the origin and all vertices are defined relative to the bottom of that branch (instead of the center of the branch which is often more typical).

Let us look at the code one section at a time.

```
HRESULT CTreeActor::AddBranchSegment( BranchNode * pNode, CTriMesh * pMesh )
{
    ULONG       i;
    USHORT      j, Index;
    D3DXMATRIX  mtxInverse, mtxRot, mtxBranch;
    D3DXVECTOR3 vecPos, vecAxis, vecRight, vecOrtho, vecNormal, vecVertexPos;
    D3DXVECTOR3 vecX, vecY, vecZ;

    // Back track until we find the beginning node for this branch
    BranchNode * pStartNode = pNode;

    while ( pStartNode->Type != BRANCH_BEGIN && pStartNode )
            pStartNode = pStartNode->Parent;

    if ( !pStartNode ) return D3DERR_INVALIDCALL;
```

In the first section of the code, we need to backtrack from the current node (which may not be a branch start node) to find the start of its branch. We will need access to this branch start node so that we can build a matrix that will transform the current node we are processing from tree space into model/branch space. As you can see in the above code, we simply start at the current node and use the pStartNode pointer to climb up through the parents of each node until we hit the start branch (a branch of type BRANCH_BEGIN). We now have the branch start node stored in pStartNode and the current node we wish to build a ring of vertices for stored in pNode.

Once we have the branch start node, we want to build a matrix for it like we did before. We perform the cross product on the right and direction vectors of the start node to get the third axis of the coordinate system. We then store the three axis vectors of the start node in a matrix along with its position.

```
    // Store / generate the vectors used to build the branch matrix
    vecX = pStartNode->Right;
    vecZ = pStartNode->Direction;
    D3DXVec3Cross( &vecY, &vecZ, &vecX );

    // Generate the frame matrix for this branch
    D3DXMatrixIdentity( &mtxBranch );
    mtxBranch._11 = vecX.x; mtxBranch._12 = vecX.y; mtxBranch._13 = vecX.z;
    mtxBranch._21 = vecY.x; mtxBranch._22 = vecY.y; mtxBranch._23 = vecY.z;
    mtxBranch._31 = vecZ.x; mtxBranch._32 = vecZ.y; mtxBranch._33 = vecZ.z;
    mtxBranch._41 = pStartNode->Position.x;
    mtxBranch._42 = pStartNode->Position.y;
    mtxBranch._43 = pStartNode->Position.z;
```

We now have a matrix (mtxBranch) that describes the position and orientation of the local axis of the branch start node relative to the origin of tree space. For example, if we place a vertex at the origin of the coordinate system and then multiplied it by this matrix, we would transform that vertex out to the position of the branch start node.

What we want is a matrix that does the exact opposite. We want a matrix that will undo the transformation of the branch start node from any vector we multiply with it. For example, if the branch start node was positioned at 100 units along the X axis and we have a vertex positioned at 110, we want a matrix that will subtract the branch start node from the position (and orientation) of the vector. If M was our desired matrix and V was our vector, V*M should equal 10. In other words, we need to know the position of the current node we are about to process as an offset from the branch start position. Likewise, we will also want to know the orientation of the current node's direction and right vectors as rotational offsets from the branch start node's direction and right vectors. Of course we have seen this all before. What we are saying here is that we want our current node defined in the local space of the branch start node. And as we learned back in Module I (and mentioned again earlier in the lesson), multiplication by the inverse of the matrix will do the trick.

```
    // Get the inverse matrix, to bring the node back into the frame's space
    D3DXMatrixInverse( &mtxInverse, NULL, &mtxBranch );
```

Now that we have this matrix what will we use it for?

We need to create a ring of vertices and place them on a plane described by the current node. This is a plane where pNode->Direction (made unit length) would describe its normal. We will see how we place the vertices on this plane in a moment, but for now just know that we will need the direction vector, the right vector and a vector orthogonal to the them both in order to slide the vertices from the node center point out to their correct positions on the plane. However, as we wish the vertices to be placed in model space and not in tree space, it stands to reason that both the current node's position and its local axis vectors (direction and right vector) which will be used to position the vertices, should also be transformed into model space (start branch relative space) prior to being used to position those vertices

In the next section of code we transform the current node's direction and right vectors into model space using the inverse matrix. We then store them in the local variables vecAxis and vecRight, respectively. We also transform the node's position into model space and store it in the vecPos local variable.

```
    // Build the axis in the frame's space
    D3DXVec3TransformNormal( &vecAxis, &pNode->Direction, &mtxInverse );
    D3DXVec3TransformNormal( &vecRight, &pNode->Right, &mtxInverse );
    D3DXVec3TransformCoord ( &vecPos, &pNode->Position, &mtxInverse );
```

To understand what we have just done take a look at Figure 12.40. The figure shows an arbitrary branch defined somewhere in the tree. Concentrate on just the first two nodes (N and N+1) which show the first two nodes of the branch as defined in tree space (prior to the transform we have just performed). Notice the green arrows at the two nodes showing the direction vector at each node prior to transformation.

In Figure 12.40, the blue lines indicate the planes defined by the nodes in tree space. The green arrows show the direction vector of each node, which essentially describes the normal of those planes. In this example, the first node in the branch is not assumed to be the root of the entire tree but is instead assumed to be a child of some other parent branch.

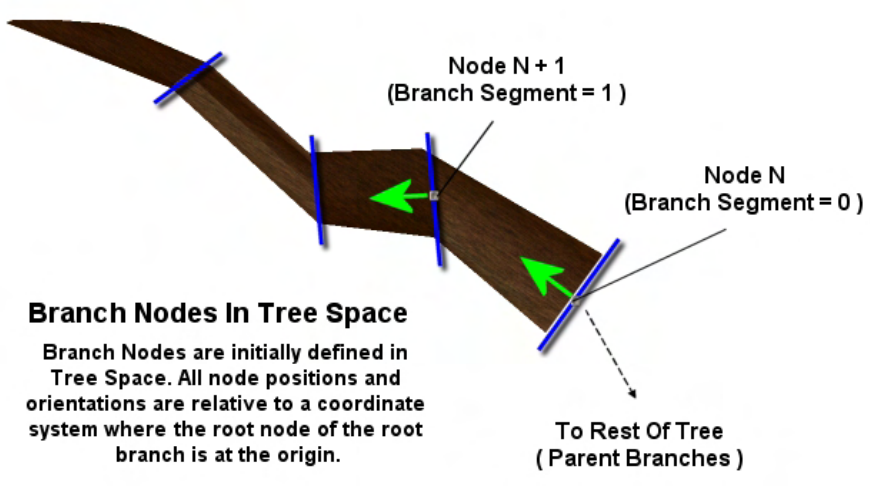


Figure 12.40 : Branch In Tree Space

In tree space, the position of each node is relative to the origin of the tree. So the position of the branch start node will not be (0,0,0) and thus is not the origin of the coordinate system. As discussed earlier, this needs to be rectified because we are going to want our children (and the vertices they represent) to be defined in the local space of the branch start node. So by applying the inverse matrix of the branch start node that we have just created to the position and orientation vectors of the current node we wish to add vertices for, we move the position and orientation of the node into the coordinate space defined by the branch start node. In Figure 12.41 you can see how all the nodes for the branch shown above look once transformed into mesh local (i.e., model) space.

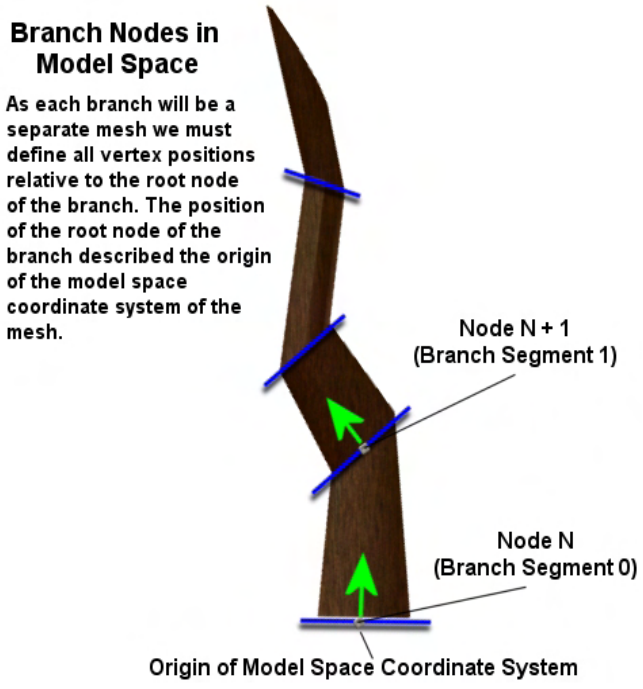


Figure 12.41 : Branch Nodes in Model Space

As you can see, in model space the nodes of the branch represent a unique mesh. The direction and right vectors of the branch start node form the Z and X axes of the coordinate system and the tree space position of the branch start node is mapped to the origin of the coordinate system.

Notice how the orientations and direction vectors in each node have changed. Yet while very different from their tree space counterparts, they still maintain the same inter-branch relationship.

It is the direction and right vector of the current node in this space which should be used to position the ring of vertices on the model space plane defined by each node. After all, when we created the start node for this branch we would have also added a frame to the actor's hierarchy. Remember, it is the frame matrices in the hierarchy that are responsible for ultimately transforming this model space branch into its proper tree space position at render time.

At this point, we now have the direction vector and the right vector of the current node in model space. The direction vector describes the normal of the plane on which the vertices should be added and the right vector is tangent to the plane (it lies on the plane). In order to position our vertices on this plane we will also need an additional tangent vector (often referred to as a binormal). If you imagine you are staring at a clock (with hands) that is currently showing 3 o'clock, think of the direction vector as an arrow coming out of the center of the clock pointing right at you. Now imagine that the little hand that is on the 3 is the right vector of the node. Finally, the binormal would be the big hand of the clock pointing at the 12 in this particular case.

If we have two tangent vectors, we have the ability to place vertices anywhere on that plane by combining those vectors and scaling. This should not surprise you since we are basically talking about a standard Cartesian XY plane here (where the direction vector is the Z axis). The X axis (1,0,0) and a Y axis (0,1,0) are both tangent vectors for the XY plane and they allow us to position vertices anywhere on that plane. If we want to place a point at position (40, 50) on the XY plane, we think of this as describing a point 40 units along the X axis and 50 units along the Y axis. However, what we are really saying is this:

Position X = 40 * (1, 0, 0) = (40, 0, 0)
+
Position Y = 50 * (0, 1, 0) = (0, 50, 0)
=
Position XY = (40, 50, 0)

This is a more mathematically correct way of thinking about what happens when we plot a coordinate. The reason we get the expected results is that our transformation matrices store an X axis and a Y axis with a length of 1.0 (unit length). But if our transformation matrix stored X and Y axis vectors with a length of 3.0 instead, the final position plotted on that plane is obviously entirely different:

Position X = 40 * (3, 0 ,0) = (120 , 0 , 0)
+
Position Y = 50 * (0, 3, 0) = (0, 150, 0)
=
Position XY = (120, 150, 0)

This is the mathematics of linear transformations and vector spaces. We do not need to get much more technical here in this course since we expect that you will cover such material in the Game Mathematics course. If you have not done so already, you should get used to viewing transformations in this way.

The point of this exercise was to demonstrate that if we have two unit length tangent vectors, we can use them to plot any point on the plane they define. When we position our vertices on the plane of the node, we will not be using the usual X and Y axis vectors as shown above. The plane of the node may be rotated and/or tilted in model space, so we will use the tangent vectors instead.

At the moment we currently have one tangent vector -- the node's right vector. We can think of this as the node's local X axis. We need a local Y axis also so that we can use these two vectors to position the ring of vertices on the plane.

Calculating this vector is easy and you should already know how to do it. The right vector is tangent to the plane and that the direction vector is perpendicular to the plane. The other tangent vector we wish to create should be orthogonal to these two, so all we have to do is rotate the model space right vector around the model space direction vector by 90 degrees and we have our second tangent vector (the binormal). This is shown in Figure 12.42.

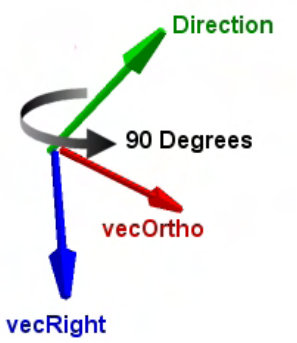


Figure 12.42

So let us build our node's third model space axis next:

```
    // Build the ortho vector which we use for our Y dimension axis
    D3DXMatrixRotationAxis( &mtxRot, &vecAxis, D3DXToRadian( 90.0f ) );
    D3DXVec3TransformNormal( &vecOrtho, &vecRight, &mtxRot );
```

We now have the node position and its local axis in model space so it is time to start adding our vertices. The second section of the function is divided into two code blocks. The first is executed only if the current node is not of type BRANCH_END. It adds a ring of vertices on the model space node plane and adds the indices to the mesh to stitch them into faces with the previous node's ring of vertices. The second code block is executed only if we are currently processing the BRANCH_END node. When this is the case, only a single vertex is added at the node position (not a ring) and the indices are added to stitch this final vertex into faces using the previous node's ring of vertices. The end result is the branch tip.

Let us first have a look at the code block that is executed for nodes that are not BRANCH_END nodes (adding rings).

```
    // If this is a beginning / segment node
    if ( pNode->Type != BRANCH_END )
    {
        // Add enough vertices for the new branch segment
        long VIndex = pMesh->AddVertex( m_Properties.Branch_Resolution );

        // Generate the vertices
        for ( i = 0; i < m_Properties.Branch_Resolution; ++i )
        {
            CVertex * pVertex = &((CVertex*)pMesh->GetVertices())[ VIndex + i ];

            // Calculate angle of rotation ((Branch_Resolution - 1)
            // because we are duplicating one vertex at the seam)
```

```
            float fAngle =
            D3DXToRadian( (360.0f / (float)(m_Properties.Branch_Resolution - 1))
                          * -(float)i );

            // Plot the points for our 'elliptical' branch hull
            float x = pNode->Dimensions.x * cosf( fAngle );
            float y = pNode->Dimensions.y * sinf( fAngle );
```

In the above code, we first tell the mesh that we are about to add a ring of vertices so that it can make room for those vertices at the end of its internal vertex arrays. We do this using the CTriMesh::AddVertex method and pass the number of new vertices we wish to make space for. The number of vertices that will be used to form a ring is stored in the Branch_Resolution member of the growth properties structure (8 by default). The AddVertex method returns the index (VIndex) of the first vertex in the array of vertices we have just added.

We then entered a loop which iterates through each vertex we wish to add and allows us to calculate the position of each new vertex in the ring one at a time. Inside the loop you can see that we retrieve a pointer to the vertex structure whose position and normal we will need to set. Notice how we use the base index of the first new vertex we allocated (VIndex) and add that onto that loop variable 'i' to allow us to step through each of the new vertices.

The next thing we did was calculate a delta angle (fAngle). If we have 8 vertices to place (e.g., Branch_Resolution = 8), then we need to step around the model space position of the node and position vertices in a circle at uniform intervals. As the first and last vertices have duplicated positions for the wrap around, we need to step around a 360 degree circle in increments of (360 / Branch_Resolution - 1). Using the default branch resolution of 8, this means for each iteration of the loop, we will need to step around the circle surrounding the node a further 360/7 = 51.42 degrees and place another vertex.

As 51.42 degrees describes the size of single wedge formed by two adjacent vertices in the ring (with the center point as the apex of the wedge), we can multiply this value with the current loop iteration variable 'i' so we know exactly where we need to place the vertex. For example, in the first three iterations of the loop, fAngle would equal the following:

Iteration 1 : 51.42*0 = 0 (Vertex placed at zero degrees about circle)
Iteration 2 : 51.42*1 = 51.42 (Vertex placed at 51.42 degrees about circle)
Iteration 3 : 51.42*2 = 102.84 (Vertex placed at 102.84 degrees about circle)

Once we have the rotation angle for the current vertex, we can calculate a position on the circumference of the ellipse with the help of the sine and cosine functions. As we know, the sine and cosine functions are 90 degrees apart, so if we wish to place a point on a circle with a radius R at rotation angle A, we calculate the X and Y coordinates of that point as:

X Position = R * cosine (A)
+
Y Position = R * sine (A)
=
(X Position , Y Position , 0)

Notice that we are ignoring the Z coordinate for the moment since we are essentially defining a 2D circle on a plane. At this particular point in time, that plane is actually the XY plane of the model space coordinate system. The circle will also be defined about the origin of the coordinate system at this point.

In our code however, the node's Dimensions.X and Dimenions.Y members define the radii of the ellipse. (If these two members are equal then we define a circle.), Since the X radius may be different from the Y radius, this turns our calculation into:

X Position = pNode->Dimensions.x * cosine (A)
+
Y Position = pNode->Dimensions.y * sine (A)
=
(X Position , Y Position , 0)

Since the cosine and sine are 90 degrees apart, they allow us to apply scaling factors to the X and Y dimensions such that the final result will always lie on the circumference of the ellipse. For example, if the angle is 0 degrees (first iteration) then the cosine will return 0 and the sign will return 1. This will result in a first vertex position of:

(pNode->Dimensions.x , 0 , 0)

The start of the circle is not the top but actually the position of the number 3 in the clock face example (along the X axis). However, at 90 degrees, the cosine returns 0 and the sine returns 1, which results in a final vector at the top of the circle:

(0 , pNode->Dimensions.y , 0)

At 45 degrees the angle is equidistant from both the X and Y axis and the result of both will be ~0.707106. In this instance we are scaling both the X and Y radii of the ellipse to produce a point that is between both axes, but still the correct distance from the circle center point (on the circumference of the circle).

If we imagine that we are calculating the position of a vertex at 45 degrees around the circle of a node that has both an x and y dimension of 1, this will result in the following X and Y coordinates:

X Position = 1 * 0.707106
+
Y Position = 1 * 0.707106
=
(0.707106 , 0.707106 , 0)

As you can see, this is exactly the same vector you would get if you normalized the vector (1, 1, 0), which we know would result in a 45 degree vector describing a position that is halfway between the X and Y axes but still has a length of one.

At the moment we are simply defining the positions of each vertex as if it was being projected onto the XY plane of the model space coordinate system. The center of the circle would be the coordinate system origin. Although we process each vertex one at a time, if we were to imagine the positions of every vertex in the ring at this point in the loop, we would see a result like Figure 12.43.

As Figure 12.43 shows, the dimensions of the current node define the radius of the ring of vertices around its center point. However, at the moment, the vertex positions are not defined relative to the node's plane. That is to say, the center of this circle is currently (0,0,0) and not the node's model space position. Furthermore, the vertices currently lay flat on the XY plane of the coordinate system and not the plane of the current node.

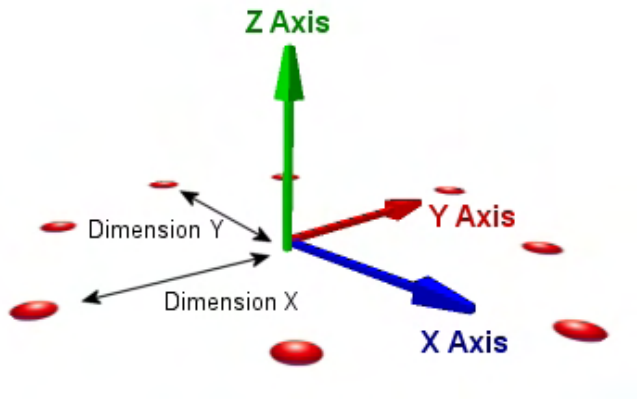


Figure 12.43 : Circle of Vertices in model space

The first thing we must do now that we have our vertex position defined with respect to the center of the ellipse is position that vertex on the node plane. The thing to remember is that the node plane may be oriented quite differently from the XY plane of the model space coordinate system. It will most likely be rotated and/or tilted in some way. So given a vertex position on the XY plane of the model space coordinate system, how do we place it in its corresponding position on another plane with a totally different orientation?

A little while ago, we said that when we specify a coordinate in a Cartesian coordinate system, we are really stating that these components should be multiplied with the X, Y, and Z axes of the coordinate system in which we are trying to plot them. Because we usually mean to plot such points in a coordinate system with the axes (1,0,0), (0,1,0) and (0,0,1), the resulting multiplication with each axis does not alter the input position of the vector. Therefore, while there was no need to perform the calculation explicitly, mathematically speaking, we calculated the position of our vertex as follows:

X Position = pNode->Dimensions.x * cosine (A)) * (1,0,0)
+
Y Position = pNode->Dimensions.y * sine (A)) * (0,1,0)
=
(X Position , Y Position , 0)

Of course, the result is the same, so there was no need to supply the additional model space axis multiplications on the end. However, Figure 12.45 shows how we might want the circle of vertices to look on the actual node plane. Remember it may be tilted or rotated with respect to the XY plane of the coordinate system shown in Figure 12.44.

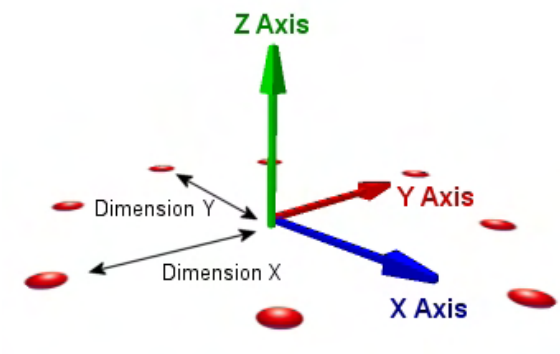


Figure 12.44 : Vertices multiplied by Model Space X and Y axis

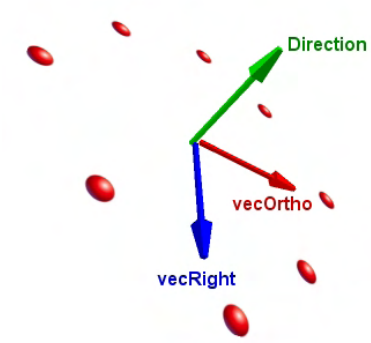


Figure 12.45 : Vertices multiplied by node tangent vectors

So let us piece together some of what we already know. When we positioned the vertices on the XY plane of the coordinate system, we were actually multiplying the X and Y position with the X and Y axes of the coordinate system shown in figure 12.44. We did not have to perform this calculation explicitly and the lack of such a calculation means that it was implied.

In Figure 12.45 we see that our node plane also has its own X and Y axes which are tangent to the plane (vecRight and vecOrtho). Therefore, all we have to do is multiply the vertex position that is currently on the model space XY plane with the node's local axes and we can move each vertex onto the plane.

```
            // Set and scale the vertex position based on the chosen dimension.
            vecVertexPos = (vecRight * x) + (vecOrtho * y);

            // Generate our normal from this position
            D3DXVec3Normalize( &vecNormal, &vecVertexPos );
```

As you can see, we multiply the node's right vector (X axis) with the vertex X position and the node's other tangent vector (the Y axis) with the vertex Y position and then sum the resulting vectors.

At this point, the plane the vertex is on is not really the node plane; it is just has the same orientation as the node plane. It still passes through the origin of the coordinate system (it has a distance of zero). In a moment we can fix this by simply adding the model space position of the node to the vertex position to move it into place on the node plane.

But before we do this, notice that in the above code we calculate a normal for the vertex simply by taking the position of the vertex and normalizing it. Remember, at this point, although we have oriented the plane (and its vertices) the origin of the coordinate system is still the center of the ellipse. Therefore, the vertex position itself represents a vector from the center of that circle out to the vertex.

Figure 12.46 illustrates this concept for a single vertex defined on the oriented plane with the origin of the coordinate system still at the center of the circle.

The black arrow pointing from the center of the circle is the vector that is represented by the vertex position at this time. If we normalize this vector, it actually serves as a fairly good vertex normal for our purposes. After the mesh is complete, if you prefer, you could do an averaging sweep through the mesh such that the vertex normal is actually the average of both the face normals to the left and right of it. Actually, most vertices will belong to four faces as it will form the top row of vertices for one cylinder and the bottom row of vertices for the next cylinder. However, we do not perform this averaging step and simply use this normal 'as is' which seems to give fairly good results when lighting the tree.

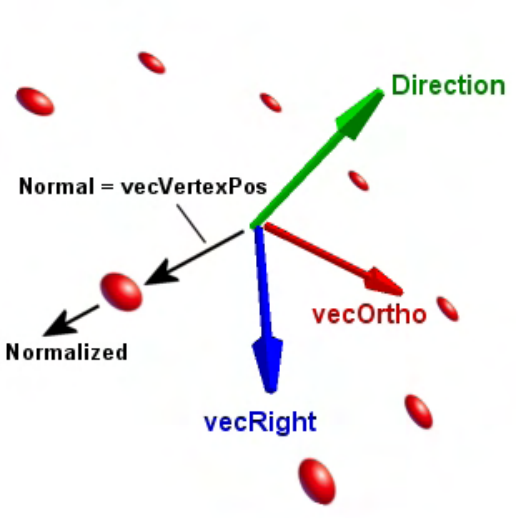


Figure 12.46 :
Normalized Vertex Pos = Vertex Normal

As mentioned, the vertex is oriented on the plane correctly but the center of the circle is still the origin of the coordinate system, not the node position. That is easily fixed by adding our current vertex position and the position of the node and storing the result (along with the normal) in our vertex structure.

```
            // Push out to it's final position, relative to the branch node
            vecVertexPos += vecPos;

            // Store the values in the vertex
            pVertex->x = vecVertexPos.x;
            pVertex->y = vecVertexPos.y;
            pVertex->z = vecVertexPos.z;

            // Store the vertex normal
            pVertex->Normal = vecNormal;
```

Before finishing this vertex we have to assign it a pair of UV texture coordinates. We discussed the technique used for UV calculation earlier in this lesson. The U coordinate is assigned a position across the width of the texture which is a product of the vertex index 'i' within the ring divided by the total number of vertices comprising a ring. This assures that vertex[0] in the ring is mapped to a U coordinate of [0] and vertex [N] is mapped to a U coordinate of 1.0 (where N is the branch resolution). All other vertices between the start and end vertices of the ring will be uniformly mapped across the width of the texture. We also scale the final result by the Texture_Scale_U growth property to allow tiling and/or texture sizing. If left at 1.0, the width of the texture will be wrapped around the cylinder/branch segment once.

Notice that we use Branch_Resolution - 1 because the first and last vertices are in the same position. We want the first vertex to have a U coordinate of 0.0 and the last vertex to have a U coordinate of 1.0 even though they are at the same position in the ring. If we do not, our mapping will be incorrect in the area between the last and first vertex. For example, if we had a ring of 10 vertices where each had its own unique position in the ring (no duplicates), the U coordinates assigned to the last two vertices would be

0.9 and 1.0. We can see that when mapping the face between the last two vertices, the texture would have its final $1/10^{th}$ (the interval 0.9 to 1.0) copied between vertices 9 and 10 in the ring. This is absolutely correct. However, when the texture is mapped to the final face in the ring between vertex 10 and vertex 1 (to wrap around) we would be mapping between a vertex with a texture coordinate of 1.0 and the first vertex with a texture coordinate of 0.0 which gives us a mapping interval of 1.0 - 0.0. In other words, the entire width of the texture (in reverse) would be mapped into the space between the last and first vertex.

By duplicating the first and last vertex positions and assigning them different texture coordinates, we can rest assured that we have two U texture coordinates at that initial vertex position in the ring. For the first two vertices in the ring in our current example the U interval would be 0.0 – 0.1 and for the last the two vertices the interval would be 0.9 - 1.0 (instead of 1.0 - 0.0 which would otherwise be the case if the first vertex was used to complete the last face).

```
            // Generate texture coordinates
            pVertex->tu = ((float)i / (float)(m_Properties.Branch_Resolution - 1))
                            * m_Properties.Texture_Scale_U;

            if ( !pNode->Parent )
                pVertex->tv = 0.0f;
            else
                pVertex->tv = ((float)(pNode->Iteration + 1) /
                               (float)m_Properties.Max_Iteration_Count)
                                    * m_Properties.Texture_Scale_V;

        } // Next Vertex
```

Notice when calculating the V coordinate, if the current node has no parent then it is simply set to 0 (the bottom/top of the texture). Remember from our earlier discussion that unlike the U coordinate, the V coordinate is not local to the ring. In fact, when the Texture_Scale_V growth property is at its default value of 1.0, the height of the texture is mapped to the entire height of the tree. Thus, its level in the virtual tree hierarchy determines the node's V texture coordinate. This is just a simple case of dividing the node's iteration (depth in the hierarchy) by the maximum iteration (maximum depth of the hierarchy) to map its iteration into the 0.0 to 1.0 range. Notice that we actually use Iteration + 1. This is because (as discussed earlier) the first two levels of the branch node hierarchy have the same iteration of zero. As the first iteration will have no parent and will always be assigned a V value of 0 in the above code, we want the second node to be 1 not 0, and the third node to be 2 not 1, and so on.

And there we have it. We have now positioned all the vertices in the ring for this node, assigned them normals and texture coordinates, and added this information to the CTriMesh.

Unfortunately, our work is still not done. We still have to add indices to the mesh also so that we form a band of triangles that join all the vertices in the ring we have just added with the vertices added by the parent node (in a previous AddSegment call). Of course, we only perform this step if the current node is not a BRANCH_BEGIN node, as this would mean we would have only added one ring/row of vertices. We need at least two rings of vertices to create a branch segment, so the following code will only be executed for the second node and beyond in a given branch. Therefore, if the current node is a BRANCH_BEGIN node, we would have nothing left to do in this function.

Let us have a look at the code block that starts to add the indices to the mesh.

```
        // If this is not the start of a new branch, add indices
        if ( pNode->Type != BRANCH_BEGIN )
        {
            ULONG FaceIndex=pMesh->AddFace((m_Properties.Branch_Resolution-1)*2);
```

The first thing we do is call the CTriMesh::AddFace method to reserve space to add indices to the mesh's indices array. Two rings of N vertices allow us to create N-1 quads because there are N-1 pairs of vertices in an N vertex ring with which to form the base of a quad. We also know that if we have two rows of vertices with 8 vertices in each, then by stepping around the ring of vertices a vertex at a time and using two vertices from each ring, we could construct 7 quads to form the hull of the cylinder segment. Furthermore, as each quad would need to be constructed from 2 triangles, we know that with a default branch resolution of 8 we would need to create 7*2=14 triangles to wrap a ring of faces around our rings of vertices and form another branch segment.

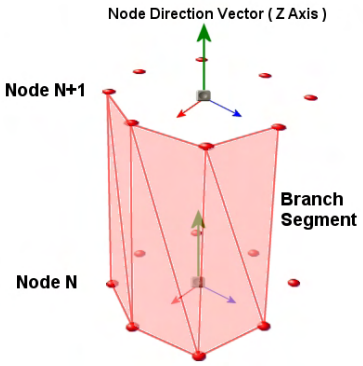


Figure 12.47

In Figure 12.47 we see two rings of vertices with a branch resolution of 8 (which actually creates 8 unique vertex positions that can be seen). While we have not filled in all the faces, you should be able to fill in the blanks yourself and see that in this case we would need to allocate space for 8*2 = 16 triangles. Node N+1 is actually the current node that we are processing and Node N is the ring of vertices that was added to the mesh when the previous/parent node was processed.

Remember that the CTriMesh::AddFace method returns the index of the first face in the array of new triangles that have been added to the mesh. We can use the CTriMesh::GetFaces method to get a pointer to the indices array and offset a pointer to the first new index we just allocated and now must populate. For example, let us assume we have just added 16 faces to the mesh (via the above call) and it returned us a face index of 80. This means that face 80 in the mesh is the first face of the 16 that we just allocated in the face array (80 have been added in previous function calls). Therefore, faces 80 through 95 will be the faces that represent the segment we are about to add.

Although we can use the CTriMesh::GetFaces method to fetch a pointer to the indices array, we only want to alter the indices used by faces 80 through 95. That is no problem since we already know the index of the first new face we added. Since all faces are triangles with three vertices, we just have to make our index pointer offset (80 * 3) into the array in this example. That is exactly why CTriMesh::GetFaces returns that useful 'first new face' index.

```
            // Retrieve the face index array
            USHORT * pIndices = &((USHORT*)pMesh->GetFaces())[FaceIndex * 3];

            USHORT Row1 = (pNode->Parent) ? pNode->Parent->VertexStart : 0;
            USHORT Row2 = (USHORT)VIndex;

            // Store this nodes vertex start
            pNode->VertexStart = Row2
```

What are the Row1 and Row2 locals doing in the above code? We need to access two rings of vertices in order to build this new branch segment. We need to know the indices for the vertices we have just added for the current node's ring as well as the indices of the parent node's ring so that we can stitch them together to make faces. Earlier in the function when we called CTriMesh::AddVertex to make room for our ring of vertices at the current node we were returned VIndex. VIndex describes the start location (first vertex) in our mesh's vertex array for the ring of vertices we have just added. This will be the second row of vertices that will form the top of the branch segment we are about to create. Notice in the above code how the node also stores the index of the first vertex used by this ring. When another branch segment is added to this branch, when we visit the child node, we can simply fetch this value from the parent node to know where its vertices start. You see this happening in the calculation of Row1 in the above code.

If the current node we are processing has no parent, then this is the first segment we have added (second node of the entire tree) and we know that the vertices for the previous node (the branch begin) must start at zero. If it does have a parent node then we fetch the index at which its vertices start in the current mesh (via its VertexStart member). As this code is only executed when we are not processing a branch start node, the VertexStart member for a branch start node will be left at its default value of zero. This is as it should be, because for any individual branch, its first ring of vertices will be created from the branch start node and will be at the very beginning of its vertex buffer.

At this point, we have an index telling us where the parent node's ring of vertices start in the vertex array and an index telling us where the current node's vertices have been added in that same array. All that is left to do now is loop around the ring of vertices and add the indices of the triangle. With each iteration of this loop we will add a quad to the index buffer (i.e., two triangles, 6 indices). Since pIndices currently points to the position in the mesh's index array where we want to start adding this index data, we can simply increment a counter variable (Index) by 6 during each iteration of the loop.

Hopefully, the following code will make sense to you (you may need to review it a few times and work through it on paper). For the first triangle of the quad we index two vertices from the previous node's ring and one vertex from the current node's ring. For the second triangle of the quad we use two vertices from current ring of vertices and one from the previous node's ring.

```
            // For each new face
            for(j=0, Index=0; j<m_Properties.Branch_Resolution-1;++j, Index+= 6 )
            {
                // Add the indices for the first triangle
                pIndices[ Index     ] = Row1 + j;
                pIndices[ Index + 1 ] = Row2 + j;
                pIndices[ Index + 2 ] = Row1 + j + 1;

                // Add the indices for the second triangle
                pIndices[ Index + 3 ] = Row2 + j;
                pIndices[ Index + 4 ] = Row2 + j + 1;
                pIndices[ Index + 5 ] = Row1 + j + 1;

            } // Next Face
        } // End if not beginning of branch
    } // End if not end node
```

As you can see, we are adding six indices (two triangles) with each iteration of the loop and using loop variable ‘j’ to step through the vertices in the top and bottom rows each time.

We have now seen all the code that is executed to add a segment when the current node is not a node of type BRANCH_END. When the current node is an end node things have to be done a little differently. We no longer add a ring of vertices at the current node; instead, we just add one vertex at the model space node position. This vertex is essentially being positioned at the center of the ring of vertices that would have been generated for this node were it not an end node.

```
    else
    {
        // Add just the one tip vertex.
        long VIndex = pMesh->AddVertex( 1 );

        CVertex * pVertex = &((CVertex*)pMesh->GetVertices())[ VIndex ];

        // Same as the node position
        vecVertexPos = vecPos;

        // Store the values in the vertex
        pVertex->x = vecVertexPos.x;
        pVertex->y = vecVertexPos.y;
        pVertex->z = vecVertexPos.z;

        // Store the vertex normal
        pVertex->Normal = vecAxis;
```

The normal calculation is also different for this vertex since we can no longer create a normal using a vector from the center of the circle to the vertex. This is obviously because the vertex is at the center of this circle. Instead, we just use the model space direction vector of the end node as calculated in the virtual tree generation process.

Generating texture coordinates for this vertex is simple also. The U coordinate is always set to 0.5 so that it will be mapped to a point exactly half way across the width of the texture (if no scaling is being used). This seems logical if you consider that this is a tip in the center of where the circle would be, and as such is halfway between both sides of that circle. The V texture coordinate is calculated the same way as before (i.e., a function of node hierarchy depth).

```
        // Generate texture coordinates
        pVertex->tu  = 0.5f;
        pVertex->tv  = ((float)(pNode->Iteration + 1) /
                        (float)m_Properties.Max_Iteration_Count)  *
                         m_Properties.Texture_Scale_V;
```

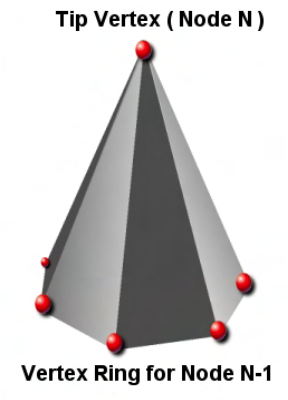


Figure 12.48

Now it is time to add the faces for this end segment. Although we added only one vertex, we still need to form a triangle between this vertex and every vertex in the ring inserted at the previous branch node. This allows us to end the branch using a cone shape (Figure 12.48).

If we have a ring of N vertices at the previous node then we know that these can be used to build the base of N-1 faces. We can see that the number of faces we wish to add to the CTriMesh in order to add this cone would be the branch resolution minus 1. As before, we will use the CTriMesh::AddFace function to allocate this many faces (essentially just multiplies the desired face count by three and allocates that many indices in the mesh's indices array).

```
        // Creating pointed tip
        ULONG FaceIndex = pMesh->AddFace( m_Properties.Branch_Resolution - 1 );

        // Retrieve the face index buffer
        USHORT * pIndices = &((USHORT*)pMesh->GetFaces())[FaceIndex * 3];

        USHORT  Row1 = (pNode->Parent) ? pNode->Parent->VertexStart : 0;
        USHORT Row2 = (USHORT)VIndex;

        // Store the vertex start
        pNode->VertexStart = Row2;
```

Notice that once again, after allocating space for the new indices, we use the CTriMesh::GetFaces function to return a pointer to the start of the index array. We then multiply the index of the first new face we have allocated by 3 to offset the indices pointer so that it points to the first new index position we must fill in.

In the following and final section of code for this function, we loop around each vertex in the ring forming a triangle with the tip vertex (Row2) and each pair of vertices from the previous ring (Row1 and Row1+1).

```
        // For each new face
        for (j=0, Index=0; j<m_Properties.Branch_Resolution - 1; ++j, Index+= 3 )
        {
            // Add the indices for the first triangle
            pIndices[ Index     ] = Row1 + j;
            pIndices[ Index + 1 ] = Row2;
            pIndices[ Index + 2 ] = Row1 + j + 1;

        } // Next Face

    } // End if end node

    // Success!!
    return D3D_OK;
}
```

We have now covered the complete AddBranchSegment function. Remember that it was called with each iteration of the BuildNode function to add a ring of vertices and indices at the current node being processed. Although on first read the function may seem quite intimidating, you should notice after further study that it really is rather intuitive (although admittedly a little long).

There is one more function we must cover before we have essentially covered the entire tree generation process. You will recall from our coverage of the BuildNode method that the BuildSkinInfo method was called whenever we were processing a branch start node (BRANCH_BEGIN). Let us have a look at this function next.

BuildSkinInfo - CTreeActor

The CTreeActor::BuildSkinInfo function is called towards the end of the BuildNode function only if the current node being processed is of type BRANCH_BEGIN. At this point in the BuildNode function, the child nodes of the BRANCH_BEGIN node will have already been visited and all the vertices and indices of each node in this branch added to the new CTriMesh.

Recall that after our CTriMesh has been built, we pass it into the CAllocateHierarchy::CreateMeshContainer function to convert it into a skinned mesh. However, this function requires that we send it an ID3DXSkinInfo object containing the information about each bone in the hierarchy that influences this mesh and the vertices in the mesh that they influence. Traditionally, we have had the ID3DXSkinInfo created for us by D3DX when loading skinned meshes from X files, but because we have hand-crafted this branch mesh ourselves, we are going to be responsible for creating the ID3DXSkinInfo object. Of course, we will also have to populate it with skinning information before passing it along with the mesh into the CreateMeshContainer function.

So what does an ID3DXSkinInfo object contain? Well, when we create a new object of this type using the global D3DX function D3DXCreateSkinInfoFVF, we must pass in the total number of bones that influence the mesh. In our case, this will be the number of bones created for nodes belonging only to this branch. We must also inform the creation function about the number of vertices in the mesh which this object will contain skinning information for.

Once this function returns, we will have an ID3DXSkinInfo object that will have an empty table of bone slots. Each row of this internal table will eventually need to store the information for one bone that influences the mesh. For example, if we have a branch that uses five bones, the empty ID3DXSkinInfo would have a table with five slots. Each element of this table will be used to store a bone name, its bone offset matrix (which we will have to calculate) and an array of vertex indices describing which vertices in the branch mesh are influenced by this bone. It will also contain an array of weights describing the strength at which the bone influences the transformation of each vertex referenced in the bone's vertex index array.

Ultimately, this function will require allocating a new ID3DXSkinInfo for N bones, and then looping N times and setting the bone name, offset matrix, vertex indices and vertex weights for each bone using the following methods of the ID3DXSkinInfo interface:

```
HRESULT SetBoneName( DWORD Bone, LPCSTR pName );

HRESULT SetBoneOffsetMatrix( DWORD Bone,
                             const D3DXMATRIX *pBoneTransform );

HRESULT SetBoneInfluence( DWORD Bone,
                          DWORD numInfluences,
                          CONST DWORD *vertices,
                          CONST FLOAT *weights );
```

The three methods shown above are the only ones we will need to populate the ID3DXSkinInfo with the information about each of the branch bones. Remember, we are only interested in the bones that influence this mesh, which is a single branch. Since the BuildSkinInfo function will only be called for BRANCH_BEGIN nodes (and passed a pointer to that node), we simply have to traverse through the child nodes of the branch, adding bones to the ID3DXSkinInfo object as we encounter them between the start and end nodes of the branch.

The SetBoneName method is used to set the name of the bone we are currently processing. The first parameter describes the zero based index of the bone along the branch and the second parameter is where we pass in a string containing the name we would like to assign to the bone. Obviously this should match the name that we gave to the actual bone/frame in the hierarchy. For example, if this is the bone at the start node of the branch, this will have an index of zero and the name passed will match the name of the frame we created for this node and attached to the hierarchy. Now you know why we stored a frame pointer in the branch node structure when we created the bone hierarchy; it allows us to easily access the bone created from a given branch node here in this function.

As you undoubtedly remember from the previous chapter, the ID3DXSkinInfo object will also store a bone offset matrix for each bone in the hierarchy. We will need to calculate the bone offset matrix ourselves. As discussed in Chapter 11, the offset matrix for a given bone is used to transform its attached vertices into the space of that bone so that local rotation of the vertices about that bone can be achieved when combined with the absolute bone matrix. The bone offset matrix is usually calculated by taking the inverse of the absolute bone matrix in its default pose. This way, when the bone matrix is rotated to some degree and is combined with the bone offset matrix, we are essentially subtracting the reference pose bone matrix from the new absolute bone matrix, leaving us with only the relative rotation to apply.

Normally, when loading a skeletal construct from an X file, the frames of the hierarchy will compose a single skeleton (a character for example). As such, the bone offset matrix for each bone is simply calculated by traversing the hierarchy from the root frame and combining matrices as we step through the levels of the hierarchy, generating the absolute (not parent relative) bone matrix for each frame. We can then invert this matrix and have a matrix that will transform that bone and its vertices into a space where the bone itself is at the origin of the coordinate system during the transformation of the vertex by that bone (i.e., bone local space.)

Things are slightly different in our tree case because our frame hierarchy does not represent the skeleton of a single mesh. Rather, our hierarchy represents multiple mini-skeletons connected together, where each mini-skeleton is the skeleton for a single branch mesh. When calculating bone offset matrices, we are only interested in transforming the bone (and any influential vertices) back to the position of the root

bone of the skeleton for the current mesh. Therefore, we do not wish the bone offset matrix we generate for a bone to be the inverse of the complete concatenation of parent-relative matrices from the root frame of the entire hierarchy right down to that bone. Instead, we wish the bone offset matrix to be only the inverse of the concatenation of matrices from the bone that begins that branch (the root bone of the branch) to the current bone being processed. In short, when calculating the bone offset matrices, we will only be interested in the skeleton of the current branch mesh and not all the bones in the entire tree. Thus, this function will traverse through the child nodes of the branch (starting at the branch start node) and at each bone it visits, concatenate the relative matrices before passing the result down to the child. For any given bone, this matrix will contain the absolute transformation of the bone in the reference pose. All we have to do is invert this matrix and we have the bone offset matrix for the current bone being processed. We can then assign it to the bone using the ID3DXSkinInfo::SetBoneOffsetMatrix method.

The next step is collecting the indices of all vertices that are influenced by the bone we are currently processing so that we can send them into the ID3DXSkinInfo::SetBoneInfluence function. This function accepts as its first parameter the index of the bone we wish to set the vertex influences for. This will be the bone we are currently processing. We also inform this function about the number of vertices it will influence, which we will count when collecting those vertices (more on this in a moment). As the final two parameters to this function we send in pointers to two arrays. The first is the array of vertex indices describing the vertices influenced by this bone. The second is an array of weight values describing how strongly the bone influences the vertices in the previous array. As you will see in a moment, all vertices are influenced by only one bone in our branch mesh, so we will always pass in an array filled with a weight value of 1.0 for each vertex.

Additionally, since no vertex will ever be influenced by more than one bone and the vertices will be added to the ID3DXSkinInfo in the same order that they were added to our CTriMesh when traversing the tree, the indices we pass into this array will always have a 1:1 mapping with the order in which the vertex data was added to the mesh. For example, if the first bone influences the first 3 rings of vertices (starting from the branch start node) and the branch resolution is 8, when setting the first bone we would pass in an array of 8*3=24 indices where the indices in this array range from 0 to 23. Similarly, if the second bone in the branch influences the following 3 rings of vertices, we will be passing another 24 indices into this array when adding the second bone's data, which goes from 24 to 47, and so on. The number of rings of vertices that are influenced by a bone is the same for all bones (except the last). This value is always Bone_Resolution*Branch_Resolution. In other words, if the bone resolution is set to 3 then we know this means that, starting from the beginning of the branch, a bone will be inserted every 3 nodes. A bone is always inserted at the branch start node. Therefore, all rings of vertices inserted from that bone node up to, but *not* including the next bone node, are considered to be influenced by that bone. If the bone resolution was 3 and the branch resolution was 8 this means that every bone would influence 3 rings of vertices (including the vertices inserted at the bone node). If the branch resolution was 8, then each bone would influence 8*3 = 24 vertices. Therefore, we would have to assign 24 vertex indices and 24 weights (all set to 1.0) to each bone we create. The exception is the final bone in the branch which may have ended prematurely by some random calculation during the tree calculation process and because the final node will contain just a single tip vertex.

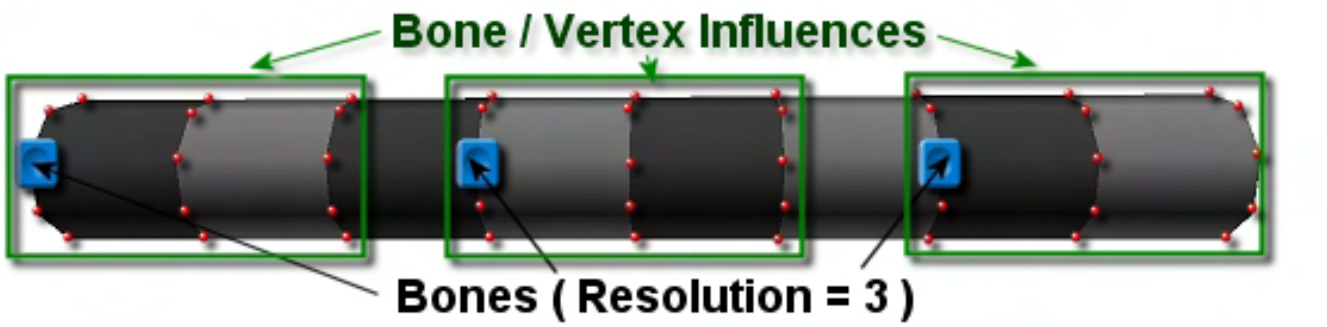


Figure 12.49 : Bone Vertex Influences for : Bone Resolution=3

Figure 12.59 shows the section of a branch where the bone resolution is set to 3. The blue boxes are the bones that are positioned every three ring's of vertices. The green boxes show the vertices which will be mapped to the bone and the bone's influence range over the branch.

Looking at this diagram and remembering that the pNode parameter passed into the BuildSkinInfo function will always be a node of type BRANCH_BEGIN, our strategy becomes clear:

1. Calculate the total number of vertices and bones in the branch.
2. Use the information from step 1 to allocate a new ID3DXSkinInfo of the correct size.
3. Step through each node in the branch starting at the branch start node.
    a. If this node is a bone node then add the name of the bone to the ID3DXSkinInfo and calculate its bone offset matrix. This will be an identity matrix for the start node of the branch.
    b. Add the indices of vertices at this node to a temporary index array.
    c. Step down into child node.
        i. If this node is a bone node, then assign all currently collected vertex indices which reference the previous bone created in step 3a, to the ID3DXSkinInfo object. Empty the temporary vertex index array.
        ii. If this is not a bone node then append the ring of vertex indices at this node to the temporary vertex index array.
4. Repeat steps 3a through 3c until last node is processed

As outlined above, we need to step through the nodes of the branch starting at the root **branch** node. When we find a bone node, we add this bone's information (name and offset matrix) to the ID3DXSkinInfo object. We then continue to traverse until the next bone is reached, collecting vertex indices in a temporary buffer along the way. As soon as we hit a new bone node, we know that the vertex indices currently stored in our temporary array belong to the previous bone, so we call the SetBoneInfluence method to assign them to that bone. We then empty the temporary vertex array so that it can be used to collect indices between the next two bones. We create a new bone at the current node and continue the process.

Let us now look at the code one section at a time. The function accepts three parameters that are passed to it by the BuildNode function. The first is a pointer to the BRANCH_BEGIN node of the branch mesh we are about to calculate the bone influences for. The second is a pointer to the branch mesh, which at this point contains all its vertex and index data. The third parameter is the address of an ID3DXSkinInfo interface pointer which on function return should point to a valid ID3DXSkinInfo interface containing

the bone influences for the mesh. The BuildNode method can then pass this interface into the CreateMeshContainer function for final skinning.

```
HRESULT CTreeActor::BuildSkinInfo( BranchNode *pNode,
                                   CTriMesh * pMeshData,
                                    LPD3DXSKININFO *ppSkinInfo )
{
    HRESULT hRet;
    ULONG  BoneCount   = 0, InfluenceCount=0, Counter=0,IndexCounter = 0, i;
    ULONG  VertexCount = pMeshData->GetNumVertices();

    ULONG      * pIndices    = NULL;
    float      * pWeights    = NULL;
    BranchNode * pSearchNode = NULL;
    BranchNode * pSegmentNode= NULL;
    TCHAR        strName[1024];
    D3DXMATRIX   mtxOffset, mtxInverse;

    // Set offset to identity.
    D3DXMatrixIdentity( &mtxOffset )
```

In the above code we calculate the number of vertices in the mesh and store it in the VertexCount parameter. We will need to know how many vertices our branch mesh contains so that we can feed it into the ID3DXCreateSkinInfoFVF function. We also initialize the local matrix variable (mtxOffset) to an identity matrix. This will be used to store the bone offset matrix for each bone we add to the ID3DXSkinInfo object. However, as the first bone we will add will be the bone at the branch start node, this is the base frame of reference for the entire mesh and the bone offset matrix should be an identity matrix.

In the next section of code we will use the pSearchNode pointer to count all the bones we have added to this branch in the hierarchy. We will need to supply D3DX with this information when it creates the ID3DXSkinInfo object.

Note that the current node may have multiple children if other branch nodes begin at the next node. That is, the next node in the current branch might be in a sibling list with multiple BRANCH_START nodes which we are *not* interesting in processing. Therefore, to locate the next child node we will step down to the first child node of the current node and then search for the one (and only) node in that list that is not a BRANCH_BEGIN node. That is the node in that sibling list that is the continuation of this branch and the next node we must process. Essentially, the next section of code steps through every node in the branch looking for bone nodes (nodes that had bones attached to them) and for each one it finds it increments the BoneCount local variable. This will allow us to count all the bones used by this mesh so that we can allocate the ID3DXSkinInfo to manage that many bones.

```
    // Navigate our way through the branch, setting bone details
    BoneCount   = 1;
    pSearchNode = pNode;

    while ( pSearchNode = pSearchNode->Child )
    {
        // We're not interested in other branches, so shift us through until
```

```
        // we find a node that is our segment, or end node.
        while ( pSearchNode->Type == BRANCH_BEGIN && pSearchNode )
                pSearchNode = pSearchNode->Sibling;

        // If we couldn't find a node, we have an invalid hierarchy
        if ( !pSearchNode ) return D3DERR_INVALIDCALL;

        // Was a frame created here?
        if ( pSearchNode->BoneNode ) BoneCount++;

    } // Next child segment
```

At the start of the above code we initially set the bone count to 1. This is because the first node is a branch start node and will always have a bone. Also, when we enter the while loop that steps through the child branch nodes, we start at the child of the BRANCH_BEGIN node. Therefore, the initial bone at the start of the branch would not be accounted for when keeping the count. Notice that with each iteration of the outer while loop, we step down one level in the hierarchy into the current node's sibling list. The inner while loop searches that sibling list for the only non-BRANCH_BEGIN node, if multiple nodes exist in that list. Essentially, the above code is just stepping down the branch each time, finding the next child node that continues the current branch, and notching the score up for each bone node that is found.

We now have everything we need to allocate a new ID3DXSkinInfo object. We know the number of bones that influence our mesh and the number of vertices in this mesh.

```
    // Create the skin info object ready for building.
    hRet = D3DXCreateSkinInfoFVF(VertexCount, VERTEX_FVF, BoneCount, ppSkinInfo );
    if (FAILED(hRet)) return hRet;

    // Get dereferenced skin info for easy access
    LPD3DXSKININFO pSkinInfo = *ppSkinInfo;
```

When the D3DXCreateSkinInfoFVF function returns, the ID3DXSkinInfo interface pointer (the final parameter) will point to a valid interface. In this case, we pass in the ID3DXSkinInfo interface pointer that was passed into the function by BuildNode so that it will be able to access it on function return. We then assign this interface pointer to a local variable for ease of access so we do not have to de-reference the original pointer each time.

At this point we have our new ID3DXSkinInfo interface but it contains no data. It has empty slots where the bone information should be, so we will spend the remainder of this function setting the information for each bone in the branch mesh.

As mentioned previously, we will need to step through the nodes of this branch and collect the indices of all vertices that are influenced by the bone we are processing. All vertices between two bones will be influenced by the previous bone (see Figure 12.49). Therefore, while stepping between two bones, we will need a temporary vertex indices buffer where we can store the data we have collected at each node as we traversed from one bone to the next. We will also need a temporary array of the same size to store the weights of each of these vertices (one weight for each vertex). We allocate these temporary buffers to be large enough to hold all the vertices in the mesh, just to be safe (e.g., if one bone in the mesh

directly influenced all its vertices, we will still be covered). The next code block allocates the vertex index array and the vertex weights array.

```
    // Allocate enough space to hold all indices and weights
    pIndices = new ULONG[ VertexCount ];
    if ( !pIndices) return E_OUTOFMEMORY;

    pWeights = new float[ VertexCount ];
    if ( !pWeights ) { delete []pIndices; return E_OUTOFMEMORY; }
```

We will step through the nodes of this branch using a while loop, but this loop will start at the child node of the branch start node. Therefore, before we enter the loop we will give the ID3DXSkinInfo object the information (vertices and weights) for the bone stored at the branch start node. We know that the branch start node will always have a bone and that the bone will always by the first bone in the mesh (index 0).

In the next section we setup the first bone in the ID3DXSkinInfo object. We build the name of this bone in the same way we built the name of the bone when we added it to the hierarchy. We must make sure that the name we add here matches the name of the bone in the frame hierarchy it represents. Once we have the name stored in a string, we set it along with the bone offset matrix for this bone (identity for the branch start bone) as shown below.

```
    // Generate root bone name (matches frame)
    _stprintf( strName, _T("Branch_%i"), pNode->UID );

    // Set it to the skin info
    pSkinInfo->SetBoneName( Counter, strName );
    pSkinInfo->SetBoneOffsetMatrix( Counter, &mtxOffset );
    Counter++;
```

The first parameter to both the SetBoneName and SetBoneOffsetMatrix functions is the index of the bone we wish to set. Since the branch start bone is the first bone in the branch it should have an index of 0. This is the initial value of the Counter variable which was initialized at the top of the function. We then increment Counter so that when we add another bone later we know this next bone will be at index 1 in the ID3DXSkinInfo object's internal bone table.

Now we need to add the indices and weights for the vertices that are influences by this bone. We will collect this information in a while loop starting from the child node of the branch start node. This means we must make sure that we add the first ring of vertices that were inserted at the branch start node so that they are not forgotten. The next section of code loops around every vertex in the first ring (Branch_Resolution) adding the index of each vertex to the pIndices temporary array and a value of 1.0 in the corresponding temporary pWeights array.

```
    // Fill in the values so far for the root of the branch
    for ( i = 0; i < m_Properties.Branch_Resolution; ++i )
    {
        pIndices[ InfluenceCount ] = IndexCounter++;
        pWeights[ InfluenceCount ] = 1.0f;
        InfluenceCount ++;
    } // Next index
```

The IndexCounter variable will start off at zero and be incremented for *every* vertex we add; it is not reset when we encounter a new bone. If our mesh has 400 vertices, then we know that the order in which we added them to the mesh is the same order in which they are assigned to their bones. Therefore, we will essentially add vertices 0-399 in exactly that order. Of course, here, we are just adding the first ring of vertices. If the branch resolution was 8 for example, we would add the values 0-7 to the indices array. When we visit the next child node and add that ring of vertices, the index counter will have not been reset, so we will be adding values 8 through 15 to the indices array, and so on.

What is InfluenceCount in the above code? In the pIndices array we are collecting the influenced vertex indices for one bone. Therefore, influence count will be reset to zero each time we start collecting vertices for a new bone. It is simply used to place the vertex index in the correct zero based location in the indices array for the current bone. For example, if a bone is influenced by 40 vertices, while collecting those vertices, influence count will count from 0 to 39. When a new bone is encountered, we will add all currently collected vertices to the ID3DXSkinInfo for the previous bone, reset the InfluenceCount back to zero, and start collecting vertex indices all over again using the same buffer for the next bone (overwriting any previous data).

At this point we have added the first bone's name and offset matrix and have collected the first ring of vertices from the BRANCH_BEGIN node and stored them in the indices buffer. We will now enter a while loop that will start at the child node and continue to make its way down to the end of the branch. It will collect the vertices from each node exactly as we did above and add them to the indices buffer. When we finally hit another bone node, we know that the indices buffer will contain all the vertex indices influenced by the previous bone, so we can add them to the ID3DXSkinInfo and attach them to the previous bone. We then flush the indices buffer, reset the influence count and start collecting vertices for this new bone, and so on.

Let us now examine the while loop which comprises the rest of the function and adds the remaining bone information for the branch.

As before, we start by stepping down into the child node of the BRANCH_BEGIN node. This is the first node we process in this loop. We then use the same inner while loop mechanism we used before to make sure that if the child node is in a sibling list with other nodes, we locate the correct node that is the continuation of the branch (not a BRANCH_START that spawns from that node).

```
    // Now we make our way through and build the skin info
    pSearchNode = pNode;
    while ( pSearchNode = pSearchNode->Child )
    {
      // Reset the segment node (we must find one at each level)
       pSegmentNode = NULL;

       // Loop through all siblings of this node
       for ( ; pSearchNode; pSearchNode = pSearchNode->Sibling )
       {
           // If this is a begin node, skip it. We aren't interested in other branches
           if ( pSearchNode->Type == BRANCH_BEGIN ) continue;

           // If this is the segment node, store it, we want to continue down from here
           if ( pSearchNode->Type == BRANCH_SEGMENT ) pSegmentNode = pSearchNode;
```

If we find any node in the sibling list that is a BRANCH_BEGIN node, we skip it and advance to its sibling. We are searching for the node that is a continuation of the current branch; the only node in the sibling list which is of type BRANCH_SEGMENT.

If we get this far then pSearchNode points at the child node (i.e., the next node we will process). We also store a copy of this node's pointer in the pSegmentNode local variable for later use.

The first thing we will do is test to see if this is a bone node. If it is, then it is time to calculate the bone offset matrix for this new bone, build its name, and set the name and bone offset matrix in the next row of the ID3DXSkinInfo bone table.

```
        // Was a frame created here?
        if ( pSearchNode->BoneNode )
        {
            // Combine the bone matrix, and inverse for the reference pose offset
            D3DXMatrixMultiply( &mtxOffset,
                                &pSearchNode->pBone->TransformationMatrix,
                                &mtxOffset );

            D3DXMatrixInverse( &mtxInverse, NULL, &mtxOffset );

            // Generate bone name (matches frame)
            _stprintf( strName, _T("Branch_%i"), pSearchNode->UID );

            // Set it to the skin info
            pSkinInfo->SetBoneName( Counter, strName );
            pSkinInfo->SetBoneOffsetMatrix( Counter, &mtxInverse );

        } // End if bone created
```

Notice that every time we encounter a bone we combine its parent relative matrix (the frame matrix) with the current contents of the mtxOffset matrix (which will be set to identity at the start node of the branch). This means that whenever we reach a bone, it will always contains the absolute transformation of that bone in its reference pose relative to the start node of the branch. As discussed earlier, inverting this matrix gives us our bone offset matrix. Also remember that when we added the first bone we incremented the Counter variable, so if this was the first bone we encountered after the branch start node, Counter would be set to 1 and we would be setting the properties of the second bone in the ID3DXSkinInfo bone table.

As discussed, when a new bone is encountered, we need to take the current vertex indices we have collected between the previous bone and this new bone and assign them as influences for the previous bone. We also have to perform this same step if the current node is a BRANCH_END node as we have reached the end of the line. In the case when the node is an end node, we also add the final tip vertex index to the index array before we assign the influences.

```
        // If we're about to switch to a new bone, or we have completed
        // this branch, fill in the PREVIOUSLY started bone's influence details
        if ( pSearchNode->BoneNode || pSearchNode->Type == BRANCH_END )
        {
            // If this is the end of the branch, just add the last influence
```

```
            if ( pSearchNode->Type == BRANCH_END )
            {
                // There is only one tip vertex on branch ends
                pIndices[ InfluenceCount ] = IndexCounter++;
                pWeights[ InfluenceCount ] = 1.0f;
                InfluenceCount++;

            } // End if end

          // Set the bone influence details
          pSkinInfo->SetBoneInfluence(Counter-1,InfluenceCount,pIndices,pWeights );

            // Reset to start again
            InfluenceCount = 0;

            // Move on to next bone
            Counter++;

        } // End if switching
```

Notice that when we call SetBoneInfluence function we use the index Counter - 1 to assign everything accumulated to the previous bone. Because Counter always contains the index of the next bone we hope to encounter, subtracting 1 from it gives us the index of the previous bone we processed. This is the bone which the vertices we have accumulated in the pIndices buffer should be assigned to. When we add the new bone and assign the collected indices to the previous bone, we reset InfluenceCount to zero to start collecting indices at the beginning of the array again for the next bone. We also increment the Counter variable so that we know the index of the next bone we encounter and which slot it should occupy in the ID3DXSkinInfo object's bone table.

The final section of code shows what happens when the node is not a bone node or branch end node. If for example, your branch had a bone resolution of 5, the code in the above code block would only be executed every 5 nodes of the branch. For all nodes in between bone nodes, the following code would be executed. It simply adds the vertex indices for the next ring of vertices to the pIndices buffer.

```
        // Add influences to the previous bone
        if ( pSearchNode->Type == BRANCH_SEGMENT )
        {
            for ( i = 0; i < m_Properties.Branch_Resolution; ++i )
            {
                pIndices[ InfluenceCount ] = IndexCounter++;
                pWeights[ InfluenceCount ] = 1.0f;
                InfluenceCount ++;

            } // Next index

        } // End if segment

       } // Next Node Sibling

       // If we couldn't find a segment node in the sibling list
       // we've reached the end of the branch
       if ( !pSegmentNode ) break;
```

```
    } // Next child segment

    // Clean up
    delete []pIndices;
    delete []pWeights;

    // Success!!
    return D3D_OK;
}
```

At the end of the function you can see that we release the temporary index and weight arrays and return.

We have now covered the entire tree generation process and program flow returns to CTreeActor::GenerateTree. If we look at the code for that function again we can see that there is only one more task that must be performed before program flow goes back to the caller (the application in our case) and the tree is considered complete.

```
HRESULT CTreeActor::GenerateTree( ULONG Options, LPDIRECT3DDEVICE9 pD3DDevice,
                                  const D3DXVECTOR3 &vecDimensions,
                                  const D3DXVECTOR3 &vecInitialDir,
                                  ULONG BranchSeed )
{
    HRESULT            hRet;
    CAllocateHierarchy Allocator( this );

    // Validate parameters
    if ( !pD3DDevice ) return D3DERR_INVALIDCALL;

    // Release previous data.
    Release();

    // Store the D3D Device here
    m_pD3DDevice = pD3DDevice;
    m_pD3DDevice->AddRef();

    // Store options
    m_nOptions = Options;

    // Generate the branches
    hRet = GenerateBranches( vecDimensions, vecInitialDir, BranchSeed );
    if (FAILED(hRet)) return hRet;

    // Build the frame hierarchy
    hRet = BuildFrameHierarchy( &Allocator );
    if (FAILED(hRet)) return hRet;

    // Build the bone matrix tables for all skinned meshes stored here
    if ( m_pFrameRoot )
    {
        hRet = BuildBoneMatrixPointers( m_pFrameRoot );
        if ( FAILED(hRet) ) return hRet;

    } // End if no hierarchy
```

```
    // All is well.
    return D3D_OK;
}
```

Just as we did in CActor::LoadActorFromX, as soon as the mesh hierarchy has been constructed we must call the base class BuildBoneMatrixPointers function. Hopefully you will recall from Lab Project 11.1 that this function traverses the entire hierarchy searching for mesh containers. For each mesh container found that stores an ID3DXSkinInfo object, it extracts each of the bone names, searches for the matching frame in the hierarchy, and adds the pointer to each frame's absolute matrix to the mesh container's bone matrix pointer array. When this function returns, each mesh container will contain an array of bone matrix pointers and an array of matching bone offset matrices that can be easily accessed and combined when rendering the mesh.

12.5 Animating CTreeActor

The CTreeActor class includes a function that creates animation data for the frame hierarchy. The GenerateAnimation function should be called by the application after the call to the GenerateTree function. This function is not automatically called by GenerateTree since you might want to generate a tree that does not animate, or perhaps you want to animate the tree using your own algorithms. Furthermore, if you intend to simply save the tree data out to disk and import it into GILES™ for placement purposes, the animation data will be lost anyway.

However, we felt that it would be helpful to go through the process of creating some simple animations. It should provide you with some additional insight into the animation system as a whole and hopefully spark some of your own creative ideas regarding animation. If you wish to use the feature, the GenerateAnimation function can be called to build keyframes for each bone in the tree. The animation we will create will simulate wind and the tree branches will sway back and forth.

To get an idea of the overall process, below we see how some code may look in an application that has created a CTreeActor and would like it to animate.

```
// Allocate new Tree Actor
CTreeActor *pTree = new CTreeActor;

// Set actor skinning mode
pTree->SetSkinningMethod( CActor::SKINMETHOD_AUTODETECT );

// Register global texture/material callback for non-managed mode
pTree->RegisterCallback( CActor::CALLBACK_ATTRIBUTEID, CollectAttributeID, this );

// Set the texture and material to use for the branches
pTree->SetBranchMaterial( _T("LondonPlaneBark.dds"), NULL );

// Generate skinned tree
pTree->GenerateTree( D3DXMESH_MANAGED,
                     m_pD3DDevice,
                     D3DXVECTOR3( 2.0f, 2.0f, 2.5f ),
```

```
                        D3DXVECTOR3( 0.0f, 1.0f, 0.0f ));

// We wish to animate so generate animation data and controller for actor
pTree->GenerateAnimation( D3DXVECTOR3( 0.0f, 0.0f, 1.0f ), 30.0f, true );
```

For completeness, we see the actor being allocated and its skinning method being set to auto-detection mode. We then see the scene registering a callback function that will be used by the actor to process the textures and materials needed by the actor. The application then calls the SetBranchMaterial method to supply the actor with the filename of the texture it would like mapped to its branches. Notice that the second material parameter, which can be used to pass a pointer to a D3DMATERIAL9 structure, is set to NULL in this example. We have elected to let the tree use the default material it creates in its constructor.

The call to the GenerateTree method is then made with the final two vectors describing the dimensions of the root node and the direction vector for that node. When this function returns, the actor will have a final frame hierarchy and every branch will be represented by a skinned mesh.

On the final line we call the CTreeActor::GenerateAnimation method, which will create the actor's animation controller and populate it with animation data. The function takes three parameters. The first is the wind direction vector which will be used to calculate how the bones of the tree will animate. This vector describes the direction that the wind is blowing and is used to calculate a rotation axis for the bones. The second vector is the strength of the wind, which is used to scale the rotation of each branch with respect to the rotation axis. The third parameter is a boolean which specifies whether the CActor::ApplyCustomSets function of the base class will be called after the sets have been created. As you will recall, this function converts each ID3DXKeyframedAnimationSet into one of our custom CAnimationSets to overcome the GetSRT bug. Since the GenerateAnimation function will only create one animation set for the entire tree, only this one animation set will have to be converted in the ApplyCustomSets method. However, we have left this parameter as a Boolean so that should this bug be fixed in the future, you can pass false and use the ID3DXKeyframedAnimationSets that are initially created. Lets us now examine the code to this function and discuss the animation it is trying to implement.

GenerateAnimation - CTreeActor

The animation we will create will be very simple. Given a wind direction vector, we will perform the cross product between that vector and the world up vector <0, 1, 0> to get a vector that is perpendicular to the wind direction vector. We can think of this new vector as being a rotation axis which the direction vector is blowing directly against. If we create a scenario where the axis exists at each bone in the tree, we can rotate the branches about this axis by a small amount determined by the fWindStrength parameter passed into the function.

By default, we will create 20 keyframes for each bone in the hierarchy. For each of those keyframes (for a given frame) we will rotate the frame by some amount. Each keyframe for every frame will contain a rotation around the same axis, but at varying degrees. The pattern we want to simulate is not simply for the branches to always be blowing away from the wind, as this would simply bend the tree away from

the wind direction vector. Rather, we wish to rotate the bones away from the wind vector but also let them move back again so that the branches sway back and forth. Admittedly, this is not particularly realistic, nor is it going to be physically correct, but the end result will be close enough to what we want to work for our purposes in this lesson.

As an example of how things will work, the animation stored in each keyframe for a given frame might slowly rotate the bone an angle of 30 degrees away from the wind vector around the rotation axis in keyframes 1 – 10. It might then spend the next 10 keyframes rotating it back again to its initial position. Our animation will not be quite that simplistic, but hopefully you understand that each frame will have an array of 20 keyframes and that at each keyframe, we store the rotation angle of the bone around the rotation axis at that particular time. To make things more realistic, we will also introduce the concept of a height delay, so that the further up the tree we move, the bones rotate slightly behind the bones beneath them in the tree. This will generate a whip-like effect, where the bottom of branches will start to move first and the end of the branch will be dragged along with it a short time later. At the midway point, the bottom section of a branch may be rotating back to its initial pose while it top part is still on its way out to full extension (the position at which the bottom section starts to pull the top section of the branch back with it).

The actual creation of the keyframe data is done using a recursive function called BuildNodeAnimation. It is called from the GenerateAnimation function to traverse all the nodes in the tree and build the keyframes for each one. We will discuss this all shortly. First though, let us have a look at the parent function and the doorway to this recursive process.

One of the first things this function must do is create the actor's ID3DXAnimationController. Because we did not load our actor's data using the D3DXLoadMeshHierarchyFromX function, our actor will currently have no animation controller assigned. However, before allocating this new controller, the function will release the previous controller interface if one exists. This is done for safety as it lets the application call this function even if the actor has previously had animation defined for it. It is possible that the application may wish to call this function again to generate animation data with a different wind direction vector or wind strength than that specified the first time the function was called.

Once we have released the controller interface we must create a new one. Recall that when we use the D3DXCreateAnimationController function to create a new controller we must pass in the limits. We only need one animation set and we may as well specify the default number of tracks (2) and the default number of key-events (30) even though we do not need them.

However, you will recall from our discussion of the animation controller in Chapter 10, that when creating one, we must know how many animation outputs (i.e., frame matrices) to make room for inside the controller. Luckily, we can use the D3DXFrameNumNamedMatrices helper function to traverse the actor's frame hierarchy and count the number of named frames. We will then reserve this many animation outputs so that we reserve enough room for every frame in the hierarchy to be manipulated by the animation controller. Furthermore, once we have created the animation controller, we still have to register each frame matrix in our hierarchy with the controller so that the controller can cache a pointer to it for animation updates. We can use the other D3DX global function D3DXFrameRegisterNamedMatrices function to automate that process too. We simply pass this function the root frame of our hierarchy and it will traverse it and register the parent-relative matrix for

each named frame as an animation output in the controller. The first section of the function that performs these tasks is shown next.

```
HRESULT CTreeActor::GenerateAnimation( D3DXVECTOR3 vecWindDir,
                                       float fWindStrength,
                                       bool bApplyCustomSets /* = true */ )
{
    HRESULT                     hRet;
    D3DXTRACK_DESC              Desc;
    ULONG                       FrameCount;
    D3DXVECTOR3                 vecWindAxis;
    LPD3DXKEYFRAMEDANIMATIONSET pAnimSet = NULL;

    // If there is already an animation controller, release it. This allows
    // the application to call this function again, specifying different
    // properties without having to rebuild the tree itself.
    if ( m_pAnimController ) m_pAnimController->Release();
    m_pAnimController = NULL;

    // Count the number of frames in our hierarchy
    FrameCount = D3DXFrameNumNamedMatrices( m_pFrameRoot );

    // Create a new animation controller
    hRet = D3DXCreateAnimationController(FrameCount,1,2,30,&m_pAnimController );
    if ( FAILED(hRet) ) return hRet;

    // Register all of our frame matrices with the animation controller
    hRet = D3DXFrameRegisterNamedMatrices( m_pFrameRoot, m_pAnimController );
    if ( FAILED(hRet) ) return hRet;
```

We now have an empty animation controller that has room for one animation set. Let us now create that animation set and call it ‘SwaySet’ (it is the only set we will use). We will work using a ticks-per-second ratio (timing resolution) of 60 for our keyframe timestamps and set the animation to loop.

```
    // Create a new animation set
    hRet = D3DXCreateKeyframedAnimationSet( _T("SwaySet"),
                                            60.0f,
                                            D3DXPLAY_LOOP,
                                            FrameCount,
                                            0,
                                            NULL,
                                            &pAnimSet );
    if ( FAILED(hRet) ) return hRet;
```

The fourth parameter in the above code is where we specify the number of animations we would like to store in this animation set. Remembering that an Animation is a collection of keyframes for a single frame in the hierarchy, we know that we wish to have an animation for every hierarchy frame. Therefore we pass in the value FrameCount, which is the total number of frames in our hierarchy as calculated earlier in the function. The fifth and sixth parameters are set to zero and null as we have no need to register callback keys. As the final parameter, we pass the address of an ID3DXKeyframedAnimationSet interface pointer which, on function return, will point to our new and empty animation set. The animation set is still not registered with the controller at this point.

We now have an animation set that we will use to store the keyframes for each frame in the hierarchy. Before we calculate those keyframes however, we need to know the axis of rotation we will use for the bones of our tree. The wind direction vector was passed into the function and describes the exact direction the wind is blowing in tree space. As such, we know that this vector is perpendicular to the axis we actually wish to rotate about. For example, imagine a player mounted on a pole in a table football game (also called Foosball in some places). The wind direction can be thought of as the vector hitting the player head on, while the pole on which the player is mounted is actually the rotation axis the player will rotate around in response to that wind. Therefore, provided our wind direction vector is not equal to the world up vector <0,1,0>, which should never realistically be the case, we can cross these two vectors to get the rotation axis by which we will rotate each bone.

```
    // To make things a little more user friendly we accept a wind direction to
    // this function, however BuildNodeAnimation requires an axis about which
    // the branches will rotate. We convert this here.
    D3DXVECTOR3 vecCross = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
    D3DXVec3Cross( &vecWindAxis, &vecCross, &vecWindDir );
```

We now have the rotation axis stored in the vecWindAxis member, so it is time to walk through the nodes of the tree and generate the keyframe data. This step is handled with a single call to the recursive function BuildNodeAnimation, whose code we will discuss next. This function is passed the root node of the branch node hierarchy, the rotation axis we just calculated, the wind strength, and the interface of our new animation set, which will be the recipient of the keyframe information calculated by this function.

```
    // Build the animations
    hRet = BuildNodeAnimation( m_pHeadNode, vecWindAxis, fWindStrength, pAnimSet);
    if ( FAILED(hRet) ) { pAnimSet->Release(); return hRet; }
```

When this function returns, every frame in the actor's hierarchy will have had its animation generated and added to the animation set. Next we must register this animation set with our animation controller and assign it to the first track on the mixer.

```
    // Register the animation set with the controller
    hRet = m_pAnimController->RegisterAnimationSet( pAnimSet );
    if ( FAILED(hRet) ) { pAnimSet->Release(); return hRet; }

    // Set this in track 0 to ensure it plays
    SetTrackAnimationSet( 0, pAnimSet );
```

We must be careful to initialize the properties of any track we use, especially when creating the animation controller manually. D3DX will not perform any track property initialization (we learned the hard way in our tests that the initial properties of a track are all zero -- which generates no visual output). In the following section of code we set the properties of the first and only mixer track we will use such that it is enabled, has a speed of 1.0 (the default), a weight of 1.0 (the default) and has a track position of 0.0 (the start).

```
    // Setup the track description (defaults to disabled)
    ZeroMemory( &Desc, sizeof(D3DXTRACK_DESC) );
```

```
    Desc.Enable   = true;
    Desc.Weight   = 1.0f;
    Desc.Speed    = 1.0f;
    Desc.Position = 0.0f;

    // Set the track description
    m_pAnimController->SetTrackDesc( 0, &Desc );
```

At this point we can release the pAnimSet set interface since it has been registered with the controller and had its reference count increased.

```
    // We can release the animation set now (controller owns it)
    pAnimSet->Release();
```

As a final step we see whether the application would like the ID3DXKeyframedAnimationSet we have just created replaced with our custom CAnimationSet object. If true is passed as this parameter, we call the base class method ApplyCustomSets, which we know from the previous lesson performs this task.

```
    // Apply custom animation sets if requested
    if ( bApplyCustomSets ) ApplyCustomSets();

    // Success!!
    return D3D_OK;

}
```

The function then returns to the application with a completely valid animation controller that can be used by the application to update the animation (via AdvanceTime).

There was nothing too challenging happening in that function, but then again, the real work is performed in the BuildNodeAnimation call. This function recursively visits every frame in the hierarchy, generating animation data for it. Let us have a look at that final function now.

BuildNodeAnimation - CTreeActor

When this function is first called, it is passed the root node of the branch node hierarchy. The function will recursively call itself visiting all child and sibling nodes in the tree. However, only significant action is taken when the node we are visiting is a bone node. If it is a bone node then we know this node has a corresponding frame in the hierarchy and it must have an animation built for it and registered with the passed animation set.

Virtually all the code in this function is placed within the “if(pNode->BoneNode)” code block. The only code that is not in that block comes at the end of the function when the function calls itself recursively if child and/or sibling nodes exist. Let us take a look at the first section of this function.

```
HRESULT CTreeActor::BuildNodeAnimation( BranchNode * pNode,
                                        const D3DXVECTOR3 & vecWindAxis,
```

```
                                        float fWindStrength,
                                        LPD3DXKEYFRAMEDANIMATIONSET pAnimSet )
{
    ULONG       i;
    HRESULT     hRet;
    D3DXVECTOR3 vecChildWindAxis = vecWindAxis;

    // If this is a bone node, build animation for us.
    if ( pNode->BoneNode )
    {
        D3DXKEY_VECTOR3    pTranslation[2];
        D3DXKEY_QUATERNION pRotation[20];
        D3DXQUATERNION     quatRotate, quatValue;

        // Transform the wind axis by this frame
        D3DXMATRIX  mtxInverse;
        D3DXMatrixInverse(  &mtxInverse,
                            NULL,
                            &pNode->pBone->TransformationMatrix );


        D3DXVec3TransformNormal(&vecChildWindAxis,&vecChildWindAxis,&mtxInverse );
```

The above code block is only executed for bone nodes. First it allocates space for two translation keyframes and 20 rotation keyframes. We have decided that each node will need 20 rotational keyframes to get good looking rotation results on screen. Although we never wish to actually translate any bone directly, unfortunately, we must always include two translation keyframes; one at the start one at the end. We do this is because the SRT data stored in the keyframes overwrites the data in the frame's transformation matrix when we call AdvanceTime. If no translation vectors are specified at all, then a translation vector of <0,0,0> will be assumed and will overwrite the actual parent-relative positional offset of the frame stored in its matrix.

For example, if we have a frame with a parent-relative matrix translation vector of <10,10,10>, we know that this offsets it from the position of the parent frame <10,10,10> units in parent space. However, when the animation has no translation or scale keyframes, <0,0,0> is used. In the case of our matrix, the <10,10,10> translation vector in the frame matrix would be overwritten each time with a translation value of <0,0,0>. This would essentially translate the child frame back to the position of the parent bone and cause a huge mess. As long as we supply at least two translation keyframes, the animation system will happily interpolate between them. The result of the interpolation will be used to replace the translation vector in the frame matrix.

Therefore, we extract the position vector stored in the frame matrix (<10,10,10> for example) and store that same position vector in a translation keyframe at the start of the animation (time 0) and a keyframe at the end of the animation (time = max time). Every time AdvanceTime is called by the application, the GetSRT function will interpolate between <10,10,10> and <10,10,10> and of course, generate the parent-relative position of <10,10,10>. Simply put, by inserting two translation keyframes at the start and end of the periodic position of the animation set, we retain the position vector of the frame in its default pose throughout the interpolation process. You will see us setting up these two translation frames in a moment.

Notice in the above code that we multiply the rotation axis (ChildWindAxis) by the inverse parent relative matrix of the current frame. This is because the rotation axis was originally defined in tree space and we need to perform the rotation in bone space. This is the same as multiplying the vector with the frame's bone offset matrix, although done slightly differently. In this case, we are multiplying it by the inverse of the parent relative matrix instead of using the inverse of the absolute matrix of the frame. However, notice the transformed rotation vector is passed by reference to its child node; therefore, instead of accumulating relative matrices and using the inverse as the bone offset matrix, we are instead accumulating the transformations applied to the wind vector from inverse parent relative matrices, which equates to the same thing. Suffice to say, at this point, we have the tree space wind vector transformed into the space of the bone's local coordinate system.

We do not want all nodes in our tree to rotate back and forth at the same time since this would look a little unnatural (like rotating a cardboard cut-out of a tree on a pivot). Instead, we would like the base of the tree to start rotating first. Then, the higher nodes in the tree will have a slight delay on them as if trying to catch up, but never quite doing so. Rather than a straight pivot motion, it causes a sweeping motion and the rotation applied to the lower branches is filtered slowly up to the branch end segments.

In order to accomplish this, we calculate a delay value that is a function of the current node's depth in the tree. By dividing the node's iteration value by the maximum iteration count of the virtual tree, we are essentially mapping the iterations of all nodes into the 0.0 to 1.0 range (much like when calculate the V texture coordinates of each vertex). So a node at the deepest level of the hierarchy (branch end nodes) is assigned a delay value of 1.0, and a node at the lowest level of the hierarchy is assigned a delay value of 0.0. Any nodes in between these hierarchy levels are mapped a value in the 0.0 to 1.0 range. Notice however, that we negate the calculation such that the branch end nodes will have a delay of –1 and the root would have a delay value of zero. Do not worry about how this is used at the moment, just be aware that the delay value describes the position of the node in the tree as a number from 0 (base of tree) to –1 (tips of branches).

```
        // Higher Up bigger animation delay causes a sweep motion
        float fDelay = -((float)pNode->Iteration /
                         (float)m_Properties.Max_Iteration_Count);
```

Not only do we need a delay such that bones positioned higher up the tree rotate slightly after bones placed towards the bottom of the tree, we also need another scalar that will be used to scale the amount of rotation we apply to a bone. If we think about a tree swaying in the wind, we know that thinner lighter branches experience much more motion than thicker heavier branches. So we can see that even if the base of the tree was swaying slightly from side to side by let us say –4 to +4 degrees, this same movement filtered up to the branch ends would result in much more movement. We might imagine the upper branches rotating between –15 and +15 degrees.

Thus, we will create a scalar in the 0.0 to 1.0 range that acts as a way to dampen the amount of rotation applied to each bone. Dampening will be a function of that node's depth in the branch node hierarchy. Once we calculate a rotation angle, it will be scaled by this value. A bone at the top of the tree would have no dampening (1.0) and the full angle value will be used in its rotation (Angle * 1.0). Bones at the base of the tree would have full dampening (Angle * 0.0). Bones in between these two extremes would be scaled appropriately. The nearer to the end of a branch a bone is located, the less dampening will

occur. We calculate this dampening value for the current node by dividing the position of the node in the hierarchy by the total depth of the hierarchy (much like above, only this time the value is not negated).

```
        // The amount of motion applied at this node. Simple calculation which
        //  increases the overall movement of the branch the higher up we get.
        float fMovement = ((float)pNode->Iteration /
                           (float)m_Properties.Max_Iteration_Count);
```

This is a simple way to calculate the dampening factor, but not necessarily the most realistic. You might decide to include other factors in your implementation (actual branch thickness for example).

It is now time to loop around and build the 20 keyframes for this node. We will set the maximum time of the animation to run for 600 ticks. In this loop, each keyframe we add will be for a time between 0 and 600 ticks (evenly spaced) and will be a rotation keyframe.

```
        // Retrieve the frame for the bone
        D3DXFRAME_MATRIX * pFrame = (D3DXFRAME_MATRIX*)pNode->pBone;

        // Generate rotation over n keys
        ULONG KeyCount = 20;
        float MaxTime  = 600.0f;

        for ( i = 0; i < KeyCount; ++i )
        {
            // Set the key time
            pRotation[i].Time = (MaxTime / (float)(KeyCount - 1)) * (float)i;

            // Retrieve initial rotation from transform matrix
            D3DXQuaternionRotationMatrix( &quatValue,
                                          &pFrame->TransformationMatrix );
```

At the start of the loop we calculate the time for the current keyframe we are adding for this node. To evenly space out our keyframes we divide the maximum time (600) by the total keycount - 1 (remember that keys are zero index based) to get a value of 31.57. This tells us we need a space of 31.57 ticks between each keyframe we add. As you can see, we then multiply this value by the current value of loop iteration variable ‘i’ (the current key we are processing) to generate the value for the current keyframe’s timestamp. For example, if we are adding the second keyframe, the timestamp is 1*31.57 = 31.57. Remember, the first keyframe would have been at position 0.0.

In the next step, we fetch the initial orientation of the frame’s parent-relative matrix as a quaternion representation using the D3DXQuaternionRotationMatrix function. Since rotation keyframes are stored as quaternions we can get the orientation of the frame in its reference pose as a quaternion and apply the rotation to that. We can then store the rotated quaternion in the keyframe to represent the orientation of the frame at this time. We know that later, the animation controller will fetch the quaternion and translation vectors for each keyframe and use them to rebuild the matrix for the frame. At this point, we need to do it the other way around. That is, we need to disassemble the matrix and store its orientation as a quaternion keyframe. In a moment, we will also apply a rotation to this quaternion so that it correctly represents the bone at the current time.

In the next three lines we will calculate how much rotation we wish to apply to the bone. It may look a little obfuscated, but just remember that all we are trying to do is find the correct rotation angle for this keyframe for the current node.

Although the next three lines of code are quite small, we will discuss each line one at a time so as we do not lose anybody along the way.

The first part of the calculation looks like this.

```
            // Calculate the angle of rotation
            float fInput = (((float)i / (float)(KeyCount - 1))
                           + fDelay) * (D3DX_PI * 2.0f);
```

When looking at the line above remember that 'i/(keycount-1)' is calculating the time of the current keyframe for this node in the 0.0 to 1.0 range. For example, we know there are 20 keyframes in total for this node, so if 'i' was currently 9, this would generate a parametric time value of 9/19 = 0.47 (approximate halfway through the 600 tick cycle). If we imagine this part in isolation, when calculated for every keyframe for this node, we are generating 20 parametric time values for each of the keyframes for the current bone. Obviously we will want each keyframe for this node to have a different rotation angle so that our branch sways backwards and forwards over time. It is intuitive then that the parametric time of the keyframe will need to play some part in its rotation angle.

We then add a Delay factor to this value. The Delay is in the range of –1.0 to 0.0 and will be closer to –1.0 the nearer to the ends of the branches the current bone we are processing is situated. Essentially, before the delay is added, we are calculating the actual time value parametrically for the current keyframe in the sequence for this bone. By adding the delay, we are effectively setting back the time (delaying) based on the position of the bone in the tree. While perhaps still not clear at this point, we have discussed how we wish the movement of bones nearer the top of the tree to seem more delayed than those near the bottom. So we are offsetting the parametric value by 0.0 to –1.0 based on the bone's position up the tree.

At this point that we have a possible range of values between –1.0 and 1.0 depending on the value of 'i' (the current keyframe being calculated for the node) and the delay (the iteration of the node in the tree). We will now map this –1 to +1 possible range of values into the –360 to +360 range by multiplying by 360 degrees ($\pi * 2$). Of course, we will work in radians, so the range of fInput after the above line will be [–6.28, +6.28].

Why would we want the position of the keyframe (with the appropriate delay added) mapped over the range of two circles (-360 to +360)? The reason is that we are going to use the shape of the cosine waveform to generate a scalar value which will be used to scale the wind strength value and the fMovement value we calculated earlier. At the top of the function, the fMovement value was calculated as a value between 0.0 and 1.0, based on the distance of the node from the base of the tree.

How does the cosine help us? To understand that, let us have a look at a graph of what the cosine function looks like if we plot its values over a large range of degrees (Figure 12.50).

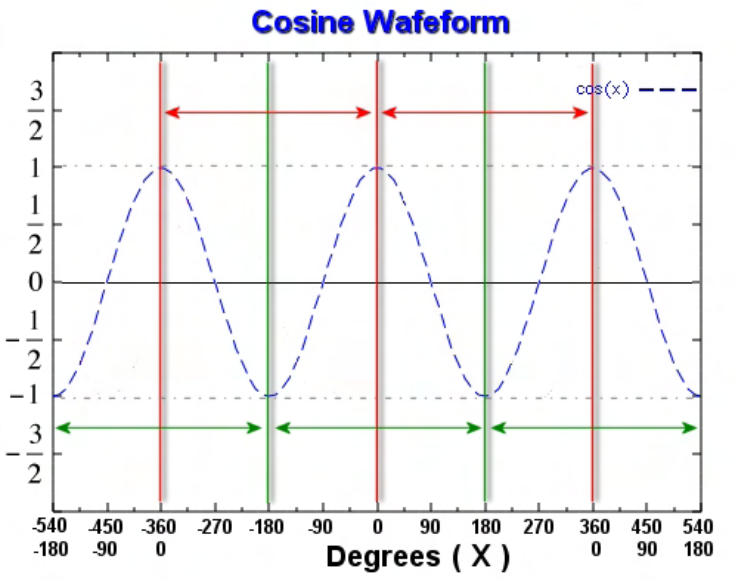


Figure 12.50

The output from the cosine function is between –1 and +1 over the range of a 360 degree circle. Furthermore, we can feed it values outside of this range and it will automatically wrap around to continue the shape of the waveform.

We can see for example that if we were to feed it values between 0 degrees (where the cosine is 1.0) to 180 degrees (where the cosine is –1) and then continue up to 360 degrees, the curve climbs back up again to a value of 1.0. This pattern is repeated as we increase or decrease values.

We can use the shape of this waveform to influence the angle by which we rotate a bone, such that the rotation pattern for the tree follows the cosine shape. For example, let us imagine that we are currently processing the first node in the tree, which would have a delay of zero. Also assume that there is a maximum of five keyframes for this frame. We know that in this case the value of fInput in the above calculation (for each keyframe) would be in the range [0, 360] degrees or [0, 6.28] radians. If we send each of these five values into the cosine function, we essentially plot five positions on the cosine waveform as shown in Figure 12.51.

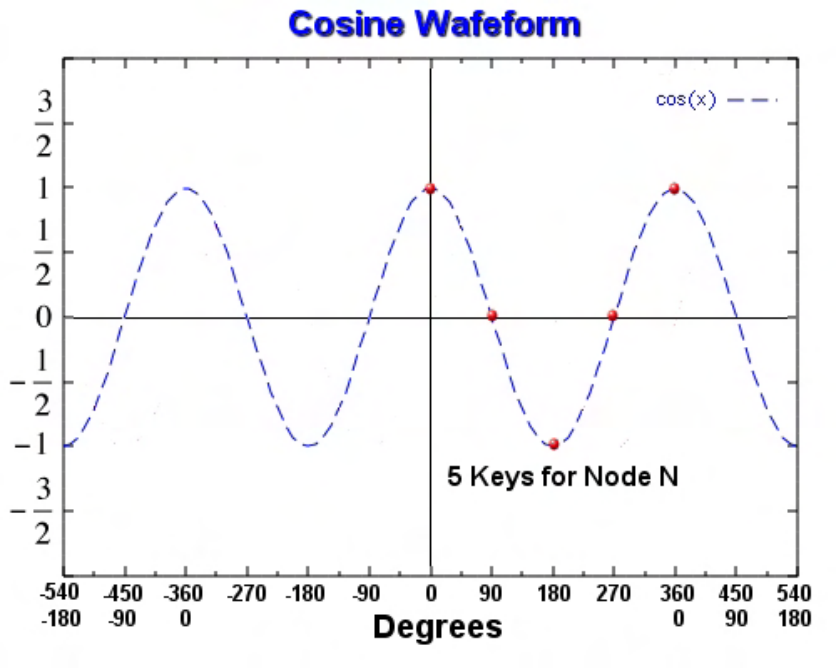


Figure 12.51

As you can see, the value assigned to each of the five nodes would be plotted on the graph and would return a value in the –1 to +1 range. This is not quite what we want at the moment, we want a value in the 0 to 1 range to act as our rotation scaling factor, but we will perform that mapping in a moment.

We see that the first node would have an input value of 0 and would therefore be assigned a height of 1 on the cosine graph. The next node would have an input of 90 degrees and would be assigned a height of 0. The third would have an input value of 180 and would be assigned a height of –180, and so on. By the time we get to the fifth node, the values start to repeat themselves.

If we imagine these values in the range of 0.0 to 1.0 (instead of –1 to +1) we can see that the result of the cosine function would return a scaling value, such that the first node would have full rotation applied, the second node half rotation, the third node no rotation, the fourth half rotation and the fifth full rotation, and the pattern continues. In other words, if we imagine these values being used to scale a rotation angle, we can see that the node would start at time zero in its fully rotated position, and over the five nodes, would rotate back to a rotation of zero and back out to a position of full rotation at the final node. Thus it is obvious why we are not happy with using the direct result from the cosine function -- it would be in the –1 to +1 range and we never want to invert the rotation angle, only scale it from zero to full rotation. The following code feeds in the input value into the cosine function and maps the result into the 0.0 to 1.0 range.

```
            float fCycle = (cosf( fInput ) + 1.0f) / 2.0f;
```

So using the above technique we essentially generate unique rotation values for each keyframe for a node. The pattern follows the shape of the cosine waveform to create a cyclical template for us to use (perfect for swaying our branches back and forth). But how did the initial fDelay value play a part? While the keyframes of the root node will always generate an fInput parameter in the range of 0-360

degrees (when fDelay=0), we must remember that the delay member can be in the range of 0.0 to –1.0. For the final node this would generate keyframes in the range of –360 to 0 degrees, which comes a whole cycle earlier on the cosine waveform. Therefore, what we are doing with this delay is moving all the nodes back from zero into the previous cosine cycle based on distance from the root node of the tree. For example, in Figure 12.52 we see how a node that is not the root node has had its cosine input value offset by the negative delay value. Note that the plotted points on the cosine graph lag behind the root node's plotted points.

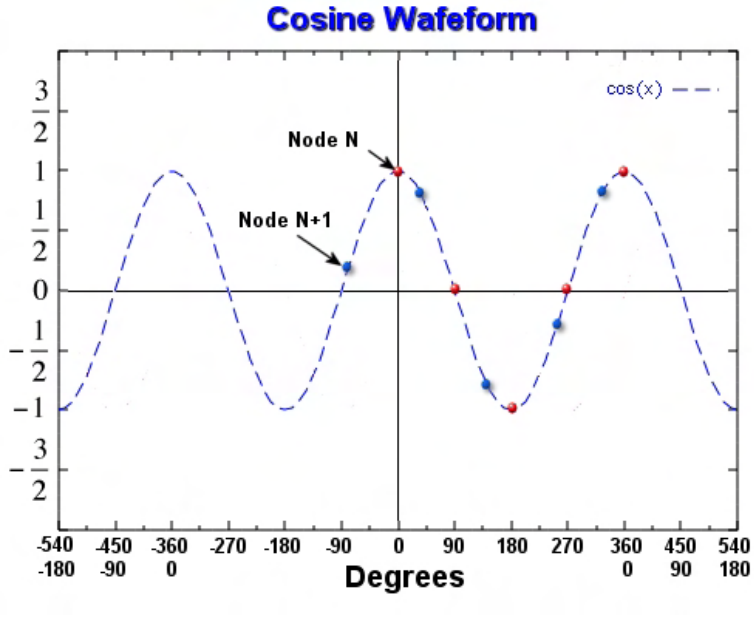


Figure 12.52

Hopefully you get the idea. The second node will have a different scaling value for each of its keyframes, but the pattern by which those angle scaling values transition from 0.0 to 1.0 will be the same. It will just be offset to some degree and mapped to a different interval of the waveform. Each node in the tree will still have the angle scaled by a complete 360 degree interval of the cosine waveform; it is just that for nodes further from the root, their initial start position on the waveform is shifted back to some position between 0 to –360.

We now have a local variable called fCycle which contains a value in the [0.0, 1.0] range. It describes how the rotation angle that we apply to this bone should be scaled based on the position of the node in the tree and the position of the keyframe we are currently processing in the keyframe list. What happens next?

Well, we know that the fWindStrength variable describes the strength of rotation in some way. Actually, we can think of this value as being the base rotation angle. For example, if a value of 30.0 was passed as the wind strength, this essentially describes a maximum rotation of 30 degrees at the branch end nodes (since the fCycle value we just calculated ranges from 0.0 to 1.0). So, we could just do fCycle * fWindStrength to get a rotation angle in the range of 0 to 30 degrees (in this example) that describes the rotation of the current keyframe. However, let us not forget that we calculated the fMovement value earlier (range = [0.0, 1.0]) which parametrically describes the distance from the

current node to the root node (0.0 = root node). As discussed, we definitely do not want the root nodes of the tree blowing back and forth nearly as much as the branch end nodes.

The fMovement value can be used to scale the wind strength angle (fMovement * fWindStrength) such that at the end nodes the rotation angle will be (1.0 * fWindStrength = Full Rotation) and at the root node it will be (0.0 * fWindStrength = No Rotation). So, we have the rotation angle for each node based on its position in the tree. This describes the maximum amount it will rotate through the sequence. We then must scale this by the fCycle value to account for the position of the current keyframe on the cosine waveform.

Angle = fCycle * (fMovement * fWindStrength)

If fWindStrength was 30 degrees and the node currently being processed was halfway up the tree from the root node, the fMovement value would be 0.5 and the maximum rotation angle for this node would be 15 degrees. Then, for each keyframe, fCycle would be a value in the range of 0.0 to 1.0, scaling the 15 degree rotation angle based on the position of the keyframe in the sequence and the delay for that node. Essentially, we generate a sequence where the bone will rotate between 0.0 and 15 degrees over its initial animation cycle and then rotate back to zero.

We are almost done, but not quite just yet. It will look unnatural if the wind is blowing our branches only to and from their default orientations. In real life, if we were to bend a branch and let it go, it would not neatly snap back to zero deviation, but likely overshoot the original center point (because of inertia) by some amount and oscillate. Perhaps it would bend 5 degrees in the negative direction before resetting itself. We will want to simulate the same thing at a simplistic level. Therefore, we will subtract 0.25 from the fCycle value to map it from [0.0, 1.0] to [–0.25, +0.75]. If a bone has a maximum rotation of 80 degrees, instead of rotating 80 degrees in one direction and then 80 degrees back to its starting position, it will actually rotate between –20 and +60 degrees. This means it will sway either side of its original starting position but with a ¾ bias in the wind direction. The final line that calculates the rotation angle is shown below.

```
  float fAngle = (fCycle - 0.25f) * (fMovement * fWindStrength);
```

We now have the angle, so our next task is to generate a rotation quaternion that represents a rotation angle equal the one we have just calculated for the wind rotation axis. To do this we will use the D3DXQuaternionRotationAxis function. We pass it an angle in radians and a rotation axis vector, and it will return a quaternion that represents the angle/axis rotation.

```
            // Rotate quaternion
            D3DXQuaternionRotationAxis( &quatRotate,
                                        &vecChildWindAxis,
                                        D3DXToRadian( fAngle ) );
```

Now all we have to do is multiply this rotation quaternion with the original quaternion we extracted from the frame matrix. This results in a quaternion that describes the orientation of the bone at this keyframe. Notice how we use the D3DXQuaternionMultiply function to rotate the original quaternion and then perform a quaternion conjugate on this quaternion to make it right handed. For some reason (as we discussed in Chapter 10), the D3DX animation system expects right handed quaternions even though

the D3DX quaternion functions generate left handed ones. The Conjugate function simply negates the axis of the quaternion so that it is pointing positive in the opposite direction (-x , -y , -z). We store the result of the conjugate directly in the keyframe's value. That completes the inner loop code and the process used to generate each of the 20 rotation keyframes for the current node.

```
            D3DXQuaternionMultiply( &quatValue, &quatValue, &quatRotate );

            // Conjugate quat into storage.
            D3DXQuaternionConjugate( &pRotation[i].Value, &quatValue );

        } // Next Key
```

As discussed earlier, we need to store at least two translation keyframes or GetSRT will interpolate between <0,0,0> and <0,0,0> (which will always return a result of <0,0,0>) and overwrite the true position of the frame when we call AdvanceTime. Therefore, for each node, we create two translation keyframes, one at the start of the timeline and one at the end, that both store the same frame position. This position is extracted from the translation vector of the frame matrix. The animation system will simply interpolate the translation value between the start position and the end position, which are the same, leaving the position of the frame constant.

```
        // Generate two bounding translation keys from original matrix
        pTranslation[0].Time  = 0.0f;
        pTranslation[0].Value = D3DXVECTOR3( pFrame->TransformationMatrix._41,
                                             pFrame->TransformationMatrix._42,
                                             pFrame->TransformationMatrix._43 );

        pTranslation[1].Time  = MaxTime;
        pTranslation[1].Value = D3DXVECTOR3( pFrame->TransformationMatrix._41,
                                             pFrame->TransformationMatrix._42,
                                             pFrame->TransformationMatrix._43 );

        // Register the animation keys
        hRet = pAnimSet->RegisterAnimationSRTKeys( pFrame->Name,
                                                   0,
                                                   KeyCount,
                                                   2,
                                                   NULL,
                                                   pRotation,
                                                   pTranslation, NULL );

    } // End if this is a bone node
```

In the above code, you can then see that after we have built the two translation vectors, we have all the information we need to build the animation for this frame in the hierarchy. We have an array of 20 rotation keyframes and 2 positional keyframes. We then register those keyframes with the animation set. Notice how we are careful to assign the name of the animation the same as the frame it is intended to animate. Remember, the animation controller uses this as its means for linking frames to animations. That ends the entire code block that is executed if the current node we are processing is a bone node. If not, all of the above code will be skipped and no animation will be created.

Finally, at the bottom of the function we see the common code that is executed for all nodes and not just bone nodes. It recurses along the sibling list if it exists and then down into the child list.

```
    // Build the nodes for child & sibling
    if ( pNode->Sibling )
    {
        hRet = BuildNodeAnimation( pNode->Sibling,
                                   vecWindAxis,
                                   fWindStrength,
                                   pAnimSet );

        if (FAILED(hRet)) return hRet;

    } // End if has sibling

    if ( pNode->Child )
    {
        hRet = BuildNodeAnimation(  pNode->Child,
                                    vecChildWindAxis,
                                    fWindStrength,
                                    pAnimSet );

        if (FAILED(hRet)) return hRet;

    } // End if has child

    // Success!!
    return D3D_OK;
}
```

When the root iteration of this function (called from GenerateAnimation) returns, the animations for every bone in the hierarchy will have been constructed and added to the animation set. We saw earlier that when program flow returns, the GenerateAnimation function then registers the animation set with the controller and assigns it to an active mixer track.

That brings us to the end of our first tree discussion. Granted, there was quite a lot to take in, but it really did provide some very good insight into how skeletons and skins and animation all tie together.

12.6 Results

Lab Project 12.1 implements everything we have discussed so far in this chapter. Our tree class does not yet support adding leaves to the branches, but that is what the second part of this chapter will discuss. An example of a tree generated by Lab Project 12.1 is shown in Figure 12.53.

Admittedly, this does not look very impressive at the moment, but when we add leaves to this tree there will be a vast improvement. Since we will essentially just be adding new code to what we have already developed, it is recommended that, before reading any further, you examine the source code to Lab Project 12.1 and make sure that you understand the previous section of this textbook.

Figure 12.53

12.7 Adding Leaves to our Trees

Figure 12.54

For the remainder of this textbook we will discuss an upgrade to our tree system that allows us to significantly increase both detail and realism. Figure 12.54 shows a screenshot of a CTreeActor generated using the techniques we will discuss in this section. Looking at the image in Figure 12.54 it would seem as if our current code is a long way from creating trees of such detail. But as it happens, all that will be needed are a few (small) extra functions and a few minor tweaks to some existing code. After that, our CTreeActor class will be generating lush green trees as illustrated here.

The process of adding leaves to our tree will be one of adding a new branch node type to the virtual tree building process. Currently, a branch node can either be a BRANCH_BEGIN node which means it starts a new branch, a BRANCH_SEGMENT node which means it is just another segment in the branch, or a BRANCH_END node which means it ends the branch and represents the insertion of the tip vertex. We also know that any node in our virtual tree can have any of these nodes as its children. For example, we know that a given branch node might have three child nodes: a segment node that continues the branch and two branch start nodes that spawn new branches from that point. We will now introduce a new node type into our virtual tree referred to as a *frond node*.

Whenever a frond node is generated, we will insert special geometry into its parent branch mesh that will create leaves and the tiny little stalks that connect those leaves to their parent branch. You will see in a moment that while this might sound complicated, we represent a frond as two, two-sided quads criss-crossed and textured with a frond texture (see Figure 12.55).

> **Note:** The botanical meaning of the word “frond” is literally a leaf and its supporting structure (its stalk). So we will be adding not just leaves, but full fronds. That is, the leaves and the tiny twigs which connect them to their parent branch.

Figure 12.55

In Figure 12.55 we see the geometry that will be added to the branch mesh for a single frond node. The texture we apply to the intersecting quads is obviously going to be very important. In our application, we use a texture that contains not both the leaves and the stalk that will attach to the parent branch. This is important because when a frond node is randomly generated along a branch during the virtual tree building process, the position assigned to the frond node will be the same as the parent node that spawned it. That is, the base of the criss-crossed polygons shown in Figure 12.55 will actually begin

inside the branch mesh (at the center of the parent branch segment). As such, the stalk of the frond will be seen sticking out of the parent branch (like little branches flourishing into leafy tips).

With regards to the virtual tree building process, frond nodes are like normal branch nodes. They have a position and a direction vector which is deviated from the parent just like normal branch nodes. There is only one real difference between a frond node and a normal branch node; a frond node will never have any child nodes of its own. We can think of them as being child branches that are always one node in length. In this way, we can think of a frond node as being a little like a child branch being spawned in the traditional sense, where its first branch node is a branch end node. Frond nodes can be spawned from any other branch node, and although it is possible to have any given branch node spawn multiple frond nodes, in our implementation we found that this was not necessary. By giving each node a chance to create a single child front node we get trees with plenty of foliage. When the frond geometry created at one branch node overlaps the frond geometry created at its neighboring branch node we get a jumbled mix of leaves that can look very full and bushy.

In Figure 12.55 we can see that by adding four quads for a given frond node (remember that both the intersecting quads must be two-faced, which equates to four quads in total) we are able to simulate a lot of detail with a very low polygon count. Of course, much depends on how detailed the frond texture is. If the texture contained a single leaf, many more frond nodes would have to be created to build up a busy looking tree (and probably would not look very good anyway). The texture image we used in our demonstration actually contains multiple leaves and their stalks, connected by a central thin branch. The frond geometry created in Figure 12.55, if added to our tree and rendered in the way illustrated, would look quite terrible because we can clearly see the black background of the texture. This makes it a little too obvious that we are just mapping textures to quads and randomly scattering them through the tree.

Figure 12.56

Not surprisingly, our frond texture has an alpha channel so that we can enable alpha testing in the pipeline and filter out the black background pixels. Only the branch and leaf pixels will be rendered to the frame buffer. This allows other branches and frond constructs to be seen through the gaps in those leaves. We will set the pipeline alpha testing mechanism to reject any pixels from the texture that are below a certain alpha value. This will stop nearly all the background pixels being rendered.

The problem with just performing this test however is that if you study the alpha channels for such textures, they usually blend from transparent to opaque around the outside of the leaves. Therefore, even with alpha testing enabled, we can still see a dark silhouette around the leaf boundaries where the pixels are slightly darker and have alpha values that are just above the rejection value set as the alpha testing reference. If we enable alpha blending, we can remedy this using the SRC_ALPHA and INV_SRCALPH blend modes from the source and destination blend modes respectively. This will cause what was once a dark silhouette around the main image to blend with the contents of the frame buffer. With alpha blending and alpha testing enabled, the boxes around the frond texture image are removed, the quads are no longer visible, and we

end up with quite organic looking foliage (see Figure 12.56). This is a very common technique in real-time games and it is used to produce all manner of foliage, not just fronds.

12.7.1 The Frond Node

Our virtual tree building function (BuildBranchNodes) will have to be modified so that at any given branch node, it also has the ability to spawn child frond nodes. It is important to realize at this point that the frond node is not really any different from a normal branch node. The node's direction and right vector will still be deviated from its parent and the direction vector of the frond node will be used in the mesh building process to construct the direction of the frond construct. Just as in the normal branch building process, where the branch node position and direction vector describe a plane on which a ring of vertices will be placed, the frond node normal and position describe the orientation of a plane on which the base of the frond quads will be mounted (Figure 12.57).

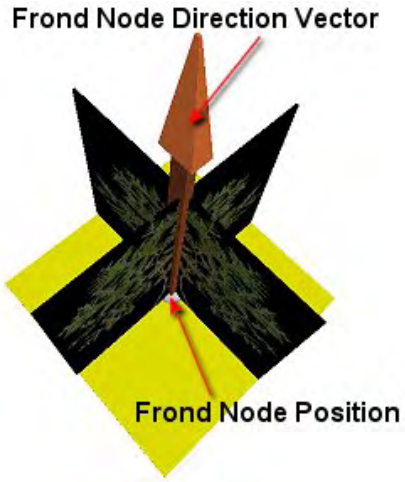


Figure 12.57

In Figure 12.57 the large brown arrow describes the direction of the node which was randomly deviated during node creation. The yellow slab represents the plane described by the node position and the direction vector. If this was a normal branch node, this the plane on which the ring of vertices would be placed. However, because this is a frond node, during the mesh building process we will build the two intersecting two-sided quads where the bottom vertices of each quad lay on the plane and the top vertices of each quad are offset from the plane in the direction of the node plane normal. That way, whichever way the plane is facing, the quads will always have their bottom vertices attached to the plane and the fronds will always point in the direction described by the node direction vector. As you have no doubt gathered by now, the only real difference between a normal branch node and a frond branch node is how each is treated during the mesh creation phase. The building of the virtual tree of branch node structures will require very few modifications. All it needs is the ability to randomly create frond nodes, just like it creates any other child node.

So that we are clear on how these nodes will be placed and how the geometry for these frond nodes will be generated, before we discuss the code let us just look at an example of how multiple frond nodes may be created to provide a nice combined visual effect.

In Figure 12.58 we see a tree that has a branch that contains a single frond node between its second and third branch segment. The position of the frond node will be inherited from its parent. Therefore, we can see that the frond node is the child of the branch node that forms the end of segment two and the beginning of segment three in the parent branch. We can also see that at the exact same spot the frond node is spawned, a new child branch is also spawned.

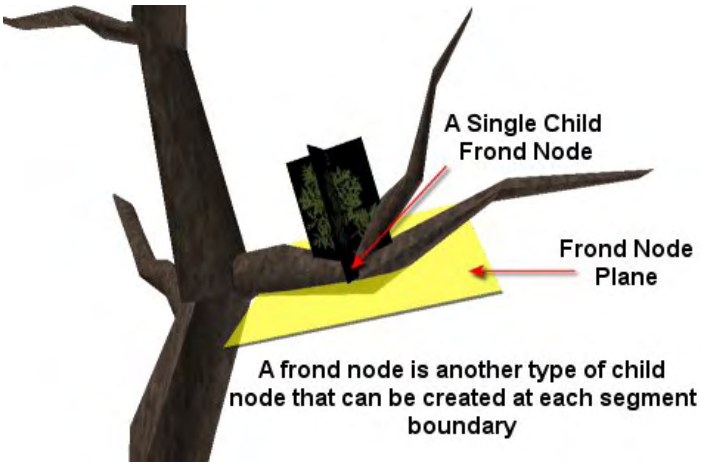


Figure 12.58

As discussed, the frond node is assigned an arbitrary direction just like any other node. The direction vector of this particular frond node, along with its position, would describe the orientation of the yellow plane in the diagram. Notice how the orientation of the frond quads is such that they are aligned to this plane. It is like the plane forms the base on which the quads are mounted and as such, the orientation of this plane determines the orientation of the frond construct. Therefore, we can think of the direction vector of the frond node as not just describing the normal to this plane, but also describing the orientation of the central spine of the frond construct where the quads intersect one another. So generating a frond node is identical to generating any other node during the virtual tree building process. We just generate a new node of type frond and assign it the direction and right vectors as normal.

The position and frequency at which frond nodes are generated will be controlled by some additional members in our TreeGrowthProperties structure. Of course, we will generally want to spawn frond nodes much more frequently than branch nodes so that we get satisfactorily dense foliage. We will often wish to spawn many child frond nodes from each parent. For example, Figure 12.59 shows the same parent node in the branch now spawning multiple frond nodes. Notice how each frond node (just like a branch start node) is randomly deviated so that we have multiple frond constructs originating from the same position in the branch but having arbitrary directions. In this example, the branch node has spawned four frond nodes as children, and also exists in a sibling list with a branch start node.

Figure 12.59

When we couple the fact that every node that meets the frond growth requirements (specified in the TreeGrowthProperties structure) can spawn multiple child frond nodes with the fact that most branch nodes will spawn child frond nodes, we have a situation where neighboring branch nodes spawn multiple fronds which all collide and overlap with each other making for a very busy looking tree. Figure 12.60 demonstrates this concept by showing just two branch nodes in close proximity that have each spawned multiple frond nodes. Even in this example, the tree is starting to look pretty busy. Imagine the same number of frond nodes being generated at most branch nodes and we can image how full our foliage will look. Because fronds from neighboring branch nodes collide and overlap it actually adds to the chaotic realism of the tree.

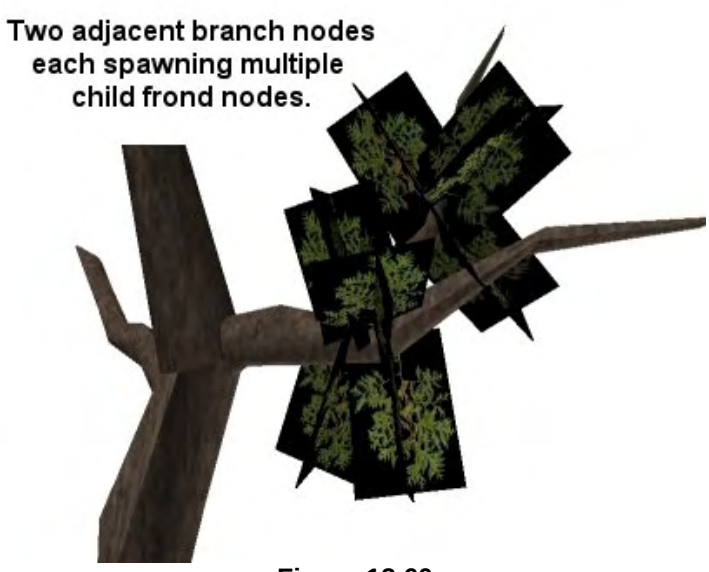


Figure 12.60

If we take the example shown in Figure 12.60 and add alpha testing and alpha blending, we can see that the results are starting to look quite pleasing even in this example where only two branch nodes are spawning fronds. The alpha tested/blended result is shown in Figure 12.61.

Figure 12.61

> **Note:** In our demo implementation we only generate one child frond node for any given parent. That is, a branch node will either have zero or one child frond nodes and never any more. However, this is certainly a behavior you can change to generate trees with more dense foliage. At the moment we are talking generically about the technique.

Now that we know exactly where frond nodes should exist and we understand the geometry they use, we can discuss the additions to the CTreeActor class that will need to be made to add frond support to the code developed in Lab Project 12.1. As we will, for the most part, simply be discussing additions to pre-existing functions, it is highly recommended that while reading this next section you open up the source

code to lab project 12.2 so that you can follow along with the code changes made in this section of the textbook. In most cases, we will not be showing all of the function code again, but only discussing the changes that have been made.

12.7.2 Updating the Virtual Tree Generation Process

As we know from the coverage of Lab Project 12.1, the first phase of the tree generation process is the construction of the virtual tree (the tree of BranchNode structures). Phase one is activated from the GenerateTree method (called by the application) via a call to the function GenerateBranches. You will recall that this function simply created the root branch node (or multiple root nodes if applicable) and then called the BuildBranchNodes function to kick off the recursive process for each root. The BuildBranchNodes function recursively called itself, adding branch nodes to the virtual tree hierarchy until it had been completely built.

For any given instance of the BuildBranchNodes method there was a simple task. The underlying question was, using the probabilities specified in the TreeGrowthProperties structure, should the current node being processed spawn a child node that continues the branch or an end node that ends it? Apart from this, it also decided how many BRANCH_BEGIN nodes should be generated as children of the current node. Recall that a BRANCH_BEGIN node signifies the beginning of a new child branch spawning from the parent.

This function will have to be modified so that it now has the option to randomly generate child frond nodes. Previously the strategy was simply:

1. Does this branch continue?
   Yes: Create child BRANCH_SEGMENT node and recur into Child Node
   No : Create child BRANCH_END node and recur into Child Node

2. Do we wish to create any child branches?
   Yes: Decide how many and create each child BRANCH_BEGIN node and recur into each
   No : Do nothing and just return

Of course, any given instance of the function returned if it had stepped into an end node. This logic will not change, but it will be supplemented. As you can see, the first step was to determine what type of node the next node in the current branch will be and to process it. The second step was to determine if child branches should also be started at this node and have the relevant child nodes created.

Now, after recursively processing the node that continues the branch and any child branch start nodes, we will add a third section that decides whether we would like to add a child frond node. Currently, each branch node can be one of three types: BRANCH_BEGIN, BRANCH_SEGMENT, or BRANCH_END. Now we will add a fourth type to the BranchNodeType enumeration called BRANCH_FROND. The updated enumeration, which is part of the CTreeActor namespace, is shown below.

Excerpt from CTreeActor.h

```
enum BranchNodeType {   BRANCH_BEGIN      = 1,
                        BRANCH_SEGMENT    = 2,
                        BRANCH_END        = 3,
                        BRANCH_FROND      = 4 };
```

Whether we create a frond node at any given parent node will once again be a random decision within limits given by the probabilities set in the TreeGrowthProperties structure. We will add five members to this structure which will be used by the virtual tree generation process to decide if a frond node should be created at a given node in the tree. Below we see the five new members of the TreeGrowthProperties structure (as defined in CTreeActor.h) followed by a description of each member and how it is used by the BuildBranchNodes method.

TreeGrowthProperties Structure (additions)

```
bool        Include_Fronds;             // Include fronds in the build
USHORT      Min_Frond_Create_Iteration; // Min iteration fronds can be created.
float       Frond_Create_Chance;        // Chance frond will be created at a node
D3DXVECTOR3 Frond_Min_Size;             // The smallest size a frond can be
D3DXVECTOR3 Frond_Max_Size;             // The largest size a frond can be
```

bool Include_Fronds

This boolean is used as a frond switch. Setting it to false will disable the generation of frond nodes and we will get trees identical to the ones from Lab Project 12.1. Setting this to true (default) will enable frond generation.

USHORT Min_Frond_Create_Iteration

We want to be able to control the node iteration at which frond nodes are created. The concept of a node's iteration is not new to us; it describes depth in the hierarchy (the number of branch segments that would need to be traversed from the root of the tree to reach that node). This value describes how many nodes deep in the hierarchy we must be before frond nodes are created. We do not usually want this to be set to a low number (or zero) or we will see frond nodes at the root of the tree. Usually, we want the fronds to start only when we get a certain way up the tree and out along the branches. The default value is 6. So fronds will not be generated at a given node while its depth in the hierarchy is smaller than 6 levels below than the root.

float Frond_Create_Chance

This property which (range = [0.0, 100.0] describes the probability that a frond node will be created as a child of the current node being processed (assuming the hierarchy depth test passes). For any given non-frond node that we are processing we will generate a random value between zero and one hundred. If the value is less than this probability, a new child frond node will be generated. Higher values for this property create more fronds and bushier trees.

D3DXVECTOR3 Frond_Min_Size
D3DXVECTOR3 Frond_Max_Size

We will usually want our fronds to become smaller in size as they progress up the tree. These two vectors describe the minimum and maximum sizes that a frond can fall between. These vectors actually describe the half-lengths of the intersecting quads (x,y) that will be generated. The z components of these vectors describe the height range for the quads (i.e., the height with respect to the node plane they

are mounted on). Larger z values will essentially create longer fronds. We will see how these values are used in the branch mesh creation phase (phase 2).

Although these properties (along with all the other tree growth properties) can be set by the application using the CTreeActor::SetGrowthProperties method, default values are specified in the constructor, as shown below.

Excerpt from CTreeActor.cpp : CTreeActor::constructor

```
// Setup frond defaults
m_Properties.Include_Fronds             = true;
m_Properties.Min_Frond_Create_Iteration = 6;
m_Properties.Frond_Create_Chance        = 25.0;
m_Properties.Frond_Min_Size             = D3DXVECTOR3( 7.5f, 7.5f, 11.25f );
m_Properties.Frond_Max_Size             = D3DXVECTOR3( 10.5f, 10.5f, 15.75f );
```

Let us now take a look at the code that has been added to the end of the BuildBranchNodes function to allow for the random generation of a child frond node.

CTreeActor::BuildBranchNodes - Updated

It is advisable to load up the source code to the BuildBranchNodes function in Lab Project 12.2. This function was covered in a lot of detail in the first section of this chapter and should be fully understood. We are now going to add a new conditional code block to the bottom of this function to create a new child BRANCH_FROND node. We will show only the new code.

```
…
…
…
} // End create child branch nodes


// Should a frond be created as a child of this node?
if ( m_Properties.Include_Fronds &&
     Iteration >= m_Properties.Min_Frond_Create_Iteration
     && (bEnd || ChanceResult( m_Properties.Frond_Create_Chance )) )
    {
```

In the above code we see that we only enter this new code block if the iteration value of the current node being processed is larger or equal to the Min_Frond_Create_Iteration growth property and if the boolean growth property bInclude_Fronds has been set to true. If these conditions are true and the ChanceResult function returns true (notice we feed in the Frond_Create_Chance growth property as the parameter) or if we have just created a child BRANCH_END node, we will create a child frond node. (We will always generate a frond node at a branch end node provided its iteration value is larger than Min_Frond_Create_Iteration).

If we get into this code block then we wish to create a new frond node. We allocate a new branch node and set its properties much like we do any other node.

```
        // If this node was the end of a branch, insert a frond node
        BranchNode * pFrondNode = new BranchNode;
        if ( !pFrondNode ) return;

        // Store node details
        pFrondNode->UID          = BranchUID++;
        pFrondNode->Parent       = pNode;
        pFrondNode->Dimensions   = pNode->Dimensions;
        pFrondNode->Position     = pNode->Position;
        pFrondNode->Direction    = pNode->Direction;
        pFrondNode->Right        = pNode->Right;
        pFrondNode->BranchSegment= 0;
        pFrondNode->Iteration    = (USHORT)Iteration + 1;
        pFrondNode->Type         = BRANCH_FROND;
```

Notice how the position and dimensions are inherited from the parent node and it is assigned its own unique ID like all other branch nodes. This time however, the Type member of the BranchNode structure is set to BRANCH_FROND. We inherit the direction and right vectors from the parent and these will be deviated in the next line of code using the same deviation angle ranges as a normal BRANCH_BEGIN node.

```
        // Deviate the node
        DeviateNode( pFrondNode,
                     m_Properties.Split_Deviation_Min_Cone,
                     m_Properties.Split_Deviation_Max_Cone,
                     m_Properties.Split_Deviation_Rotate );
```

At this point we have created a new frond node and have it located at the position of the parent node, pointing in an arbitrary direction (just like the start of a new branch). However, the frond node is not yet attached to the parent node's child list.

We need to make sure we add it to the right place in the child list if the parent node already has child nodes. It is important when building the skinning information that if the parent node contains a branch end node child, that the frond node be placed before it in the list. The vertices of the frond will be added as influences of the previous bone that was created back from it along the branch, so we wish to add these vertices before hitting the branch end node, where we add the single tip vertex and abort any further processing of the branch. The next section of code makes sure it is added to the child list prior to any BRANCH_END node that may exist there.

```
        // Link the node
        if ( !pNode->Child )
        {
            // No child already exists, just store
            pNode->Child = pFrondNode;

        } // End if no child node
```

As the above code shows, if the parent node has no other children, then we just assign its child pointer to point at the new frond node. Our new frond node will be the only node in the child list. Otherwise, the next code block is executed.

The following code sets up a while loop to traverse the child list and breaks from the loop as soon as a BRANCH_END node is found. Because with every iteration of the loop it stores the current child in its pPrev pointer before stepping into the next sibling, if the loop breaks because a BRANCH_END node is found, the pPrev pointer will point at the sibling node that exists prior to the end node in the list (the node to which we wish to add our new frond node as a sibling). That is, we will sandwich our new frond node between the branch end node and the node that existed prior to it in the sibling list.

```
        else
        {
            BranchNode * pTemp = pNode->Child, *pPrev = NULL;
            while ( pTemp )
            {
                // Have we found the end of the branch?
                if ( pTemp->Type == BRANCH_END ) break;
                pPrev = pTemp;
                pTemp = pTemp->Sibling;

            } // Next sibling

            if ( !pPrev )
            {
                // The BRANCH_END is right at the start.
                pFrondNode->Sibling = pNode->Child;
                pNode->Child = pFrondNode;

            } // End if no previous item
            else
            {
                // Attach to the 'previous' item.
                pFrondNode->Sibling = pPrev->Sibling;
                pPrev->Sibling = pFrondNode;

            } // End if previous item exists

        } // End if child node

    } // End if create frond

} // End function
```

And that represents all the code changes that had to be made in BuildBranchNode.

What is important to notice in the above code is that unlike normal branch node creation, where after it is created we recur into that node, for frond nodes we do not do this. We can think of a frond node as being a single node branch. We never recur into a frond node or add other child nodes to it.

12.7.3 Setting the Frond Texture and Material

In our previous incarnation of CTreeActor we had the ability to store the texture and material that would be used to render the branches of the tree. We exposed a method to allow the application to set these properties prior to the call to the GenerateTree function. We will now need to add two new members to CTreeActor that will contain the texture filename and material that will be used for the fronds. An accompanying member function will allow the application to set these properties prior to tree generation.

Excerpt from CTreeActor (Additional Member Variables)

```
D3DMATERIAL9 m_FrondMaterial;    // The material to apply to the fronds.
LPTSTR       m_strFrondTexture;  // The texture to apply to the fronds.
```

CTreeActor – SetFrondMaterial

This function accepts two parameters: a string containing the name of the image file we would like to use as the texture for the fronds and a pointer to a D3DMATERIAL9 structure. The texture and material are copied into the class member variables.

```
void CTreeActor::SetFrondMaterial( LPCTSTR strTexture, D3DMATERIAL9 * pMaterial )
{
    // Free any previous texture name
    if ( m_strFrondTexture ) free( m_strFrondTexture );
    m_strFrondTexture = NULL;

    // Store the material
    if ( pMaterial ) m_FrondMaterial = *pMaterial;

    // Duplicate the texture filename if any
    if ( strTexture ) m_strFrondTexture = _tcsdup( strTexture );
}
```

In the next section we will discuss the upgrades to the CTreeActor::BuildNode method. We discovered in the first section of this textbook that this method implements the second phase where the actor's hierarchy and branch skins are constructed. We will only examine the portions containing the updated code, so it is recommended that you open Lab Project 12.2 and follow along in the next section using the actual source code.

Updating CTreeActor::BuildNode

This function is called once by the BuildFrameHierarchy function and is passed the root node of the virtual tree. It steps through the hierarchy recursively from the root node visiting each branch node in the tree. For a given node, this function would add a frame to the actor's hierarchy if it was determined that

the current node should generate a bone. If the node is a BRANCH_BEGIN node, a new CTriMesh would be created and the branch would be traversed to add the vertex and index data. For any non-branch node that is not a BRANCH_BEGIN node the passed mesh is the mesh that is currently being built for the current branch being processed (i.e., the mesh that the vertices generated at this node should be added to).

In the first section of the code shown below you can see that we have snipped out nearly all the logic that generates the new frame if this is a bone node. A bone is never generated for a BRANCH_END node and likewise, if it is a BRANCH_FROND node, a bone will also *not* be created. Therefore, the code inside this conditional is completely unchanged (the conditional code never gets executed if the current node is a frond node).

We have removed all the code that generates the new frame and attaches it to the hierarchy as well as the code that creates the new CTriMesh if the current node is a BRANCH_BEGIN (bones are always added to the start of a branch). What we have left in place as a reminder at the bottom of the code block is the call to the AddBranchSegment method which was responsible for adding the ring of vertices and their indices to the mesh.

```
HRESULT CTreeActor::BuildNode( BranchNode * pNode,
                               D3DXFRAME * pParent,
                               CTriMesh * pMesh,
                               const D3DXMATRIX & mtxCombined,
                               ID3DXAllocateHierarchy * pAllocate )
{
    HRESULT     hRet;
    CTriMesh   *pNewMesh  = NULL, *pChildMesh  = NULL;
    LPD3DXFRAME pNewFrame = NULL,  pChildFrame = NULL;
    D3DXMATRIX  mtxBranch, mtxInverse, mtxChild;
    D3DXVECTOR3 vecX, vecY, vecZ;
    TCHAR       strName[1024];

    // What type of node is this
    bool bIgnoreNodeForBone = (pNode->Type == BRANCH_END ||
                               pNode->Type == BRANCH_FROND);

    if ( pNode->Type == BRANCH_BEGIN ||
        ((pNode->BranchSegment % m_Properties.Bone_Resolution) == 0 &&
         !bIgnoreNodeForBone) )
    {
        …
        …
        …
        …  --- snip ---   ( Create Frame and Setup Parameters to pass to child)
        …
        …
        …

        // Add the ring for this segment in this frame's combined space
        hRet = AddBranchSegment( pNode, pChildMesh );
        if ( FAILED(hRet) ) { if (pNewMesh) delete pNewMesh; return hRet; }

    } // End if adding a new frame
```

The next section of code was executed if the current node is not a BRANCH_BEGIN node and is not a node that should have a bone created for it. You will recall that originally this just meant calling the AddBranchSegment function to add a ring of vertices to the current mesh being built (the one passed into the function from the parent). Now we have some conditional logic. If the current node is a BRANCH_FROND node then we do not call the AddBranchSegment function since we do not wish a ring of vertices to be inserted on the frond node plane. Instead, we wish to insert vertices into the mesh that will build the intersecting quads for the frond. Therefore, we call a new method called AddBranchFrond to add the frond geometry to the current mesh. This function will be discussed in a moment.

```
    else
    {
        // Store which bone we're assigned to in the node
        pNode->pBone = pParent;

         // Is this a frond?
        if ( pNode->Type != BRANCH_FROND )
        {
            // Add the ring for this segment in this frame's combined space
            hRet = AddBranchSegment( pNode, pMesh );
            if ( FAILED(hRet) ) { return hRet; }

        } // End if not frond
        else
        {
            // Add the frond data in this frame's combined space
            hRet = AddBranchFrond( pNode, pMesh );
            if ( FAILED(hRet) ) { return hRet; }

        } // End if frond

        // Since no new frame is generated here, the child will receive
        // the same frame, mesh and matrices that we were passed.
        pChildFrame = pParent;
        pChildMesh  = pMesh;
        mtxChild    = mtxCombined;

    } // End if not creating a new frame
```

Because we only add fronds when we are not processing a bone node you might conclude that there would be gaps in the foliage whenever a branch node is a bone node. However, we must remember that this function also visits the sibling list of the bone node, so fronds can still exist at the same position as a bone node and can exist in the same list of siblings as any node type. At this point we are outside any conditional code block and back in the common flow of the function.

The next section of code steps into the child and sibling lists (this is not new code).

```
    // Build the nodes for child & sibling
    if ( pNode->Sibling )
    {
        hRet = BuildNode( pNode->Sibling,
                          pParent,
```

```
                        pMesh,
                        mtxCombined,
                        pAllocate );

        if (FAILED(hRet)) { if ( pNewMesh ) delete pNewMesh; return hRet; }

    } // End if has sibling

    if ( pNode->Child )
    {
        hRet = BuildNode( pNode->Child,
                        pChildFrame,
                        pChildMesh,
                        mtxChild,
                        pAllocate );

        if (FAILED(hRet)) { if ( pNewMesh ) delete pNewMesh; return hRet; }

    } // End if has child
```

The final section of code is executed at the bottom of the function when the current node being processed is a BRANCH_BEGIN node. Remember, because the child list is traversed before we get to this point in the function, if this is a BRANCH_BEGIN node, all the vertices in that child branch will have been added to the mesh at this point. What we have to do in this next code block is build the CTriMesh's underlying ID3DXMesh, build its ID3DXSkinInfo object, and pass this information into the CAllocateHierarchy::CreateMeshContainer function for skinned mesh creation. Virtually all of this code is unchanged, but we will comment after the first altered portion.

```
    if ( pNode->Type == BRANCH_BEGIN )
    {
        D3DXMATERIAL        Materials[2];
        D3DXMESHDATA        MeshData;
        D3DXMESHCONTAINER * pNewContainer       = NULL;
        DWORD             * pAdjacency          = NULL;
        LPD3DXBUFFER        pAdjacencyBuffer    = NULL;
        LPD3DXSKININFO      pSkinInfo           = NULL;
        ULONG               MaterialCount = ( m_Properties.Include_Fronds ) ? 2 : 1;
```

Notice the new line added to the bottom of the variable declarations. We will have to pass into the 'CreateMeshContainer' function the number of materials/subset in the mesh. If fronds are enabled we will have two subsets/materials in the mesh, so we perform this calculation here. The first subset will contain the branch faces and the second the fronds.

```
        // Generate mesh containers name
        _stprintf( strName, _T("Mesh_%i"), pNode->UID );

        // Generate the skin info for this branch
        hRet = BuildSkinInfo( pNode, pNewMesh, &pSkinInfo );

        if ( FAILED(hRet) ) { delete pNewMesh; return hRet; }

        // Signal that CTriMesh should now build the mesh in software.
```

```
        pNewMesh->BuildMesh( D3DXMESH_MANAGED, m_pD3DDevice );

        // Build the mesh data structure
        ZeroMemory( &MeshData, sizeof(D3DXMESHDATA) );
        MeshData.Type  = D3DXMESHTYPE_MESH;

        // Store a reference to our build mesh.
        // Note: This will call AddRef on the mesh itself.
        MeshData.pMesh = pNewMesh->GetMesh();

        // Build material data for this tree
        Materials[0].pTextureFilename = m_strTexture;
        Materials[0].MatD3D           = m_Material;
        Materials[1].pTextureFilename = m_strFrondTexture;
        Materials[1].MatD3D           = m_FrondMaterial;
```

The call to the BuildSkinInfo function is not changed although the contents of this function have been modified slightly, as we will see in a moment. The BuildSkinInfo function has also been upgraded so that any frond vertices also get influenced by the bones of the tree. The two new lines of code in the above code snippet are the bolded ones at the very bottom.

You will recall that when we pass our new branch mesh into the CreateMeshContainer function we must also pass in an array of materials (texture and material combinations). Originally, we just passed in a single structure since we only had one texture and material used by the entire tree. Now we have two because we also have a texture and material that will be used for the fronds (the second subset). So as you can see, we store both of these in a two element D3DXMATERIAL array before handing it off to CreateMeshContainer. Notice that we pass in the material count we calculated earlier so that CreateMeshContainer knows how many materials we are passing in the array.

```
        // Retrieve adjacency information
        pNewMesh->GenerateAdjacency( );
        pAdjacencyBuffer = pNewMesh->GetAdjacencyBuffer();
        pAdjacency       = (DWORD*)pAdjacencyBuffer->GetBufferPointer();

        // Create the new mesh container
        hRet = pAllocate->CreateMeshContainer( strName,
                                               &MeshData,
                                               Materials,
                                               NULL,
                                               MaterialCount,
                                               pAdjacency,
                                               pSkinInfo,
                                               &pNewContainer );

        // Release adjacency buffer
        pAdjacencyBuffer->Release();

        // Release the mesh we referenced
        MeshData.pMesh->Release();

        // Release the skin info
        if (pSkinInfo) pSkinInfo->Release();
```

```
        // Destroy our temporary child mesh
        delete pNewMesh;

        // If the mesh container creation failed, bail!
        if ( FAILED(hRet) ) return hRet;
```

The final change to this function requires some explanation. We will need to override the DrawActor and DrawActorSubset methods of the base class in CTreeActor so that we can enable alpha testing/blending when rendering the frond subset (subset 1). However, if our actor is in managed mode, the CreateMeshContainer function will have remapped the face attribute IDs to use the global IDs issued by the CScene class during the creation of the skin. We currently have no way of knowing what the frond subset ID will be after the CreateMeshContainer function has altered its attribute buffer. Furthermore, we cannot even lock the attribute buffer and try and determine this because the mesh may have also been attribute sorted (optimized) so we cannot even be sure that the first subset ID in the attribute buffer is the ID of the non-frond subset.

We decided to work around this problem by making a very small tweak to our CAllocateHierarchy::CreateMeshContainer function. In the next Module in this series we will examine a more comprehensive solution to this problem, but for now, our current strategy will be a lot easier to deal with.

You will recall that during the attribute remapping of a non-managed mesh we would build a temporary remap array describing what the original attribute IDs were mapped to. We then used this array to remap the attribute buffer and then the array was discarded. We will no longer discard it. Instead we store the attribute remap array in a new CAllocateHierarchy member variable so that it can be accessed. We added a small function to the CAllocateHierarchy class called GetAttributeRemap. This function will return the new ID in the remap array for an original subset ID passed in. In the following code you can see us using this new function to retrieve the new subset ID for the frond subset (originally subset 1) and storing it in a new CTreeActor member variable called m_nFrondAttribute. Our overridden DrawActorSubset subset function can then access this attribute ID and only enable the alpha blending/testing render states if the frond subset is the one that has been requested to be rendered.

```
        // Store the final attribute ID (as it was remapped) of the frond data
        if ( m_Properties.Include_Fronds )
        m_nFrondAttribute = ((CAllocateHierarchy*)pAllocate)->GetAttributeRemap( 1 );

        // Store the new mesh container in the frame
        pNewFrame->pMeshContainer = pNewContainer;

    } // End if beginning of branch

    // Success!!
    return D3D_OK;

}
```

If you followed along in the last section with your source code project open you will have seen that the changes to the code are very small indeed. However, there is now a new function called from the

BuildNode function which adds the vertex data to the mesh for a frond node. This function is called AddBranchFrond and is a brand new function whose code we will cover next.

CTreeActor - AddBranchFrond

The AddBranchFrond function is called by BuildNode whenever a frond node is encountered during phase two of the tree building process. Just as the AddBranchSegment function is called to add a ring of vertices on the node plane of a normal branch node, the AddBranchFrond function is called to add the frond vertices. This function will create two intersecting quads mounted on the node plane and aligned with the node direction vector (see Figure 12.62).

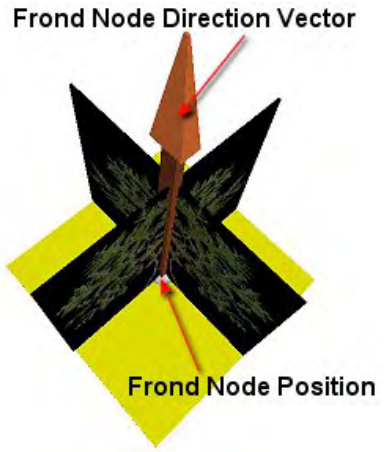


Figure 12.62

Just as we did in the AddBranchSegment function, we must transform the node position and the node plane's direction, right, and ortho vectors from tree space into branch space. This is so the vertices we generate will be relative to the beginning of the branch (the mesh in which they are contained) and not the root of the entire tree. The process we use to perform this back transformation is the same. The function is passed the frond node that needs to have its vertices added and a pointer to the CTriMesh to which the vertices should be added (the current branch mesh being built). It first executes a while loop that starts at the current node and steps back through the parent list until the BRANCH_BEGIN node is encountered. This will be the first node in the branch and the space in which we wish to transform our frond node. That is, we wish to define our frond vertices in a space where the BRANCH_BEGIN node's position is located at (0,0,0) in the coordinate system and its right, ortho and direction vectors are aligned with the X,Y, and Z axes of that system, respectively.

```
HRESULT CTreeActor::AddBranchFrond( BranchNode * pNode, CTriMesh * pMesh )
{
    USHORT      Index;
    D3DXMATRIX  mtxInverse, mtxRot, mtxBranch;
    D3DXVECTOR3 vecPos, vecAxis, vecRight, vecOrtho, vecNormal, vecVertexPos;
    D3DXVECTOR3 vecX, vecY, vecZ;
    float       fScale, fX, fY, fZ;

    // Back track until we find the beginning node for this branch
    BranchNode * pStartNode = pNode;
    while ( pStartNode->Type != BRANCH_BEGIN && pStartNode )
            pStartNode = pStartNode->Parent;

    if ( !pStartNode ) return D3DERR_INVALIDCALL;
```

At this point we now have a pointer to the BRANCH_BEGIN node, so let us extract its direction vector (it local Z axis) and its right vector (its local X axis) and perform a cross product to generate the third axis in its local system (its local Y axis). Using these three vectors we then build a transformation matrix for the branch start node and take its inverse. This gives us a matrix that will transform any position or vector into branch space.

```
    // Store / generate the vectors used to build the branch matrix
    vecX = pStartNode->Right;
    vecZ = pStartNode->Direction;
    D3DXVec3Cross( &vecY, &vecZ, &vecX );

    // Generate the frame matrix for this branch
    D3DXMatrixIdentity( &mtxBranch );
    mtxBranch._11 = vecX.x; mtxBranch._12 = vecX.y; mtxBranch._13 = vecX.z;
    mtxBranch._21 = vecY.x; mtxBranch._22 = vecY.y; mtxBranch._23 = vecY.z;
    mtxBranch._31 = vecZ.x; mtxBranch._32 = vecZ.y; mtxBranch._33 = vecZ.z;
    mtxBranch._41 = pStartNode->Position.x;
    mtxBranch._42 = pStartNode->Position.y;
    mtxBranch._43 = pStartNode->Position.z;

    // Get the inverse matrix, to bring the node back into the frame's space
    D3DXMatrixInverse( &mtxInverse, NULL, &mtxBranch );
```

Now that we have a matrix that will transform vectors into the coordinate system of the branch, we will transform the direction vector, the right vector, and the position of the frond node into branch space by multiplying them with this matrix.

```
    // Build the axis in the frame's space
    D3DXVec3TransformNormal( &vecAxis, &pNode->Direction, &mtxInverse );
    D3DXVec3TransformNormal( &vecRight, &pNode->Right, &mtxInverse );
    D3DXVec3TransformCoord ( &vecPos, &pNode->Position, &mtxInverse );
```

At this point we only have the frond node's branch space position, look vector, and right vector. In order to position the vertices on the plane, we also need the second tangent vector (the plane's Up vector). We calculate this simply by rotating the branch space right vector around the direction vector (perpendicular to the plane) by 90 degrees.

```
    // Build the ortho vector which we use for our Y dimension axis
    D3DXMatrixRotationAxis( &mtxRot, &vecAxis, D3DXToRadian( 90.0f ) );
    D3DXVec3TransformNormal( &vecOrtho, &vecRight, &mtxRot );
```

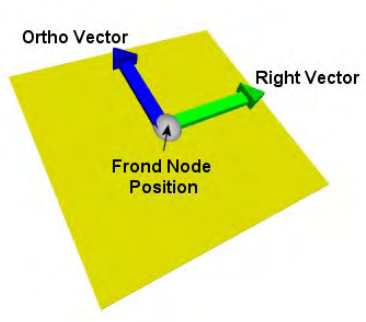


Figure 12.63

Figure 12.63 shows the plane and the two tangent vectors. Actually, because of the way we rotated our ortho vector it is actually pointing down and not up as shown in Figure 12.63, but hopefully you get the idea. As discussed in the AddBranchSegment function, we can combine the node position and the right and ortho vectors together with some scaling to position points anywhere on this plane. For example, we know that if we add the right vector to the node position, the resulting vector will be a point on the plane pointed to by the tip of the right vector in the diagram. If we add the ortho vector to the node position we would describe a position pointed to by the tip of the ortho vector in the diagram. If we add the negated right to the node position we would generate a point on the left side of the plane and similarly, if we add the negated ortho vector to the node position we would get a position at the bottom of the yellow plane shown in Figure 12.63. Those four position calculations we have just discussed describe the positions of four vertices on the

plane in the shape of a perfect cross. That is exactly where we want to base vertices of our intersecting quads to be. However, we do yet know how big we want these quads to be, so we will create some scaling values that will be multiplied with the ortho and right vectors so that we can create a larger or smaller cross on that plane.

In the next section of code we create three scaling values (fX, fY, and fZ) which will be used to scale the right, ortho and direction vectors before they are combined to create vertex positions. We will want the size of the fronds to get smaller as they are positioned higher up the tree and belong to smaller and thinner branches. We defined the minimum and maximum frond sizes as tree growth properties so we wish to find a size for the frond that is within this range but still related to the size of its parent branch. After all, a huge frond sticking out of a tiny parent branch would look unrealistic and quite bad. The technique we use is described next.

We will take the dimensions of the frond node (this was inherited from the parent when the frond node was created) and divide by the dimensions of the root node of the tree. As we know, the root node is the node in the tree with the largest dimensions, so this will create a value in the 0.0 to 1.0 range. The value is closer to zero the higher up the tree the frond is placed and the smaller its parent branch is in relation to the root node's dimensions. We will then use this scaling value to generate a new value between the minimum and maximum frond size growth properties that have been set. This is done on a per component level, so if we imagine that the variable ScaleValue describes the frond node's X dimension divided by the root node X dimension, fX would be calculated as:

fX = MinFrondSize + ((MaxFrondSize – MinFrondSize) * ScaleValue)

This is a basic interpolation calculation that we have used many times before. We are essentially adding the range (MaxFrondSize-MinFrondSize) of values that a frond can be set to, to the minimum frond size. This gives a value between MinFrondSize and MaxFrondSize based on ScaleValue (which is closer to zero the further from the root it is positioned). fX will then be used to scale the right vector of the frond node before it is added to the node position to generate the left and right vertex positions on the plane.

Below we see the code that calculates fX, fY and fZ.

```
    // If this is a frond node
    if ( pNode->Type == BRANCH_FROND )
    {
        // Calculate frond dimensions
        fScale = (pNode->Dimensions.x / m_pHeadNode->Dimensions.x);
        fX     = m_Properties.Frond_Min_Size.x +
                 ((m_Properties.Frond_Max_Size.x - m_Properties.Frond_Min_Size.x)
                 * fScale);

        fScale = (pNode->Dimensions.y / m_pHeadNode->Dimensions.y);
        fY     = m_Properties.Frond_Min_Size.y +
                  ((m_Properties.Frond_Max_Size.y - m_Properties.Frond_Min_Size.y)
                   * fScale);

        fScale = ((pNode->Dimensions.x / m_pHeadNode->Dimensions.x) +
                   (pNode->Dimensions.y / m_pHeadNode->Dimensions.y)) / 2.0f;
        fZ     = m_Properties.Frond_Min_Size.z +
```

```
                ((m_Properties.Frond_Max_Size.z - m_Properties.Frond_Min_Size.z)
                * fScale);
```

Notice that the scaling value used to create fZ is not calculated in the same way. That is, we do not scale the Z dimension of the frond node by the Z dimension of the root node. This is because, the fZ value will be used to control the length of the frond (how high the top of the quads are with respect to the node plane they are mounted on). We do not want this to be a product of the parent node's segment length but rather an average of the difference in the X and Y dimensions with respect to the root node. We would not want to have a really long frond branch if its X and Y dimensions are very small. This would create a skinny frond with a squashed texture.

Now that we have our Direction, Ortho, and Right vectors, our scaling values (fX,fY,fZ), and the node position, we can start building the vertex positions for those quads. We actually need to create four quads because we wish each of the polygons to be visible from both sides. Therefore, we will create four vertex positions per quad for a total of 16 vertex positions. Each quad will be formed from 4 vertices where two of the vertices lay on the plane and two are offset from the plane by some amount along the direction vector. The quads will be assigned their vertices such that they have a clockwise winding when viewed from the front.

In the next line of code we create a local array of 16 CVertex structures. In each element of the array we call the CVertex constructor that takes four parameters:

CVertex(D3DXVECTOR3 &vecPos, const D3DXVECTOR3 &vecNormal, float ftu, float ftv)

CVertex is defined in CObject.h and as you can see the first parameter is where we pass a 3D vector describing its position. The second parameter is where we pass in the normal of the vertex and the last two float parameters is where we pass in the texture coordinates. We will cover the first four vertex positions we add to this array one at a time so we can see how the quad is constructed. Here is the first section of the statement that defines the CVertex array and passes the first vertex.

```
// Calculate the actual frond vertices.
CVertex pVertices[16]={ CVertex( (vecPos - (vecRight*fX)), vecOrtho, 0.0f, 1.0f ),
```

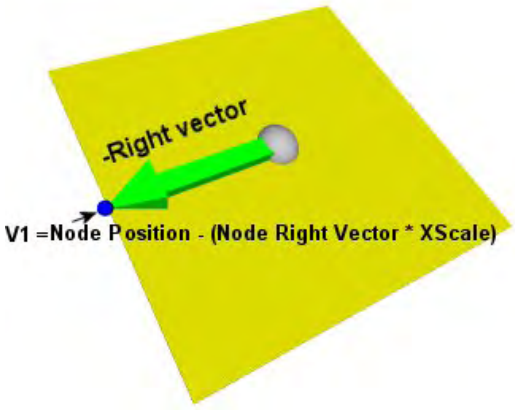


Figure 12.64

The position of the first vertex in the first quad is calculated by subtracting from the node position (shown as the ball in the center of the plane in Figure 12.64) the right vector scaled by the fX scaling value we calculated earlier. Essentially, fX dictates the width of the quad we are creating.

The resulting vertex position is situated on the plane and is shown as the small blue sphere labeled V1 in Figure 12.64. This is the bottom left corner of our first quad.

Notice that we pass in the vecOrtho vector as the vertex normal. Why? Because the right vector and the ortho vector are perpendicular to each other and therefore, in this diagram we can imagine that the ortho vector is actually pointing down towards the bottom edge of the yellow plane. This is the exact direction our quad

will be facing and thus its normal. Finally, since this is the bottom left vertex of our quad, we assign it the bottom left UV coordinate of the texture (0,1).

Let us see how we define the next (second) vertex in the array declaration.

```
CVertex( (vecPos - (vecRight * fX)) + (vecAxis * fZ), vecOrtho, 0.0f, 0.0f ),
```

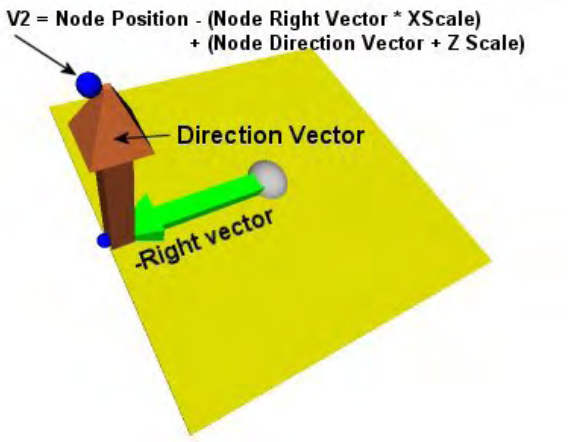


Figure 12.65

Here you can see that the next vertex position we define is initially calculated in the same way as before (by subtracting a scaled right vector from the node position) only this time we add to the resulting position the scaled direction vector (vecAxis). This vector is perpendicular to the plane and as such describes a position that is offset from the plane in the direction of the plane normal. This forms the top left corner of our quad. The same normal is used for every vertex in this quad.

As this is the top left vertex in the quad we assign it the texture coordinate for the top left of the texture (0,0).

It should be clear that in order to generate the top right and bottom right vertices we essentially do the same thing, only this time adding the right vector to the node position instead of subtracting it. This will form two symmetrical vertex positions on the right side of the node position.

```
CVertex( (vecPos + (vecRight * fX)) + (vecAxis * fZ), vecOrtho, 1.0f, 0.0f ),
CVertex( (vecPos + (vecRight * fX))                 , vecOrtho, 1.0f, 1.0f ),
```

Above we can see how the final two vertex positions are defined for the quad. These two calculations are identical to the previous two only with the addition of the right vector to the node position instead of its subtraction.

Note that it is the top right vertex we add third and the bottom right we add fourth, so that the four vertex positions of our quad describe a clockwise winding when viewed from the front.

Also notice that the two vertices are provided with the same vertex normal -- the ortho vector which is pointing toward us in this diagram. You should be able to see that the top right vertex's texture coordinate of (1,0) maps it to the top right of the texture image and the bottom left texture coordinate of (1,1) maps vertex V4 to the bottom right corner of the texture image.

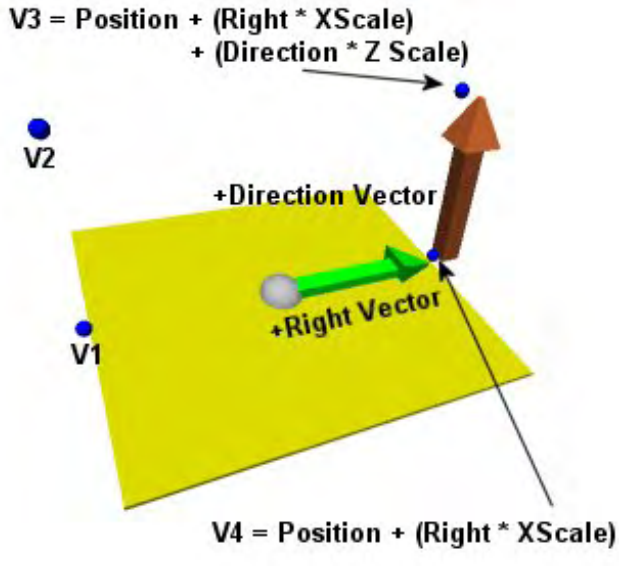


Figure 12.66

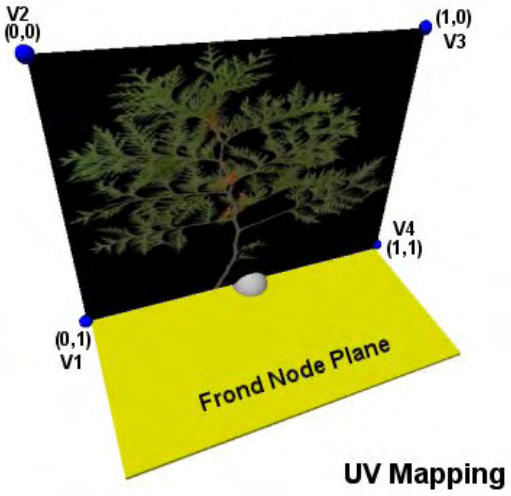


Figure 12.67

The next four vertices we define in the array describe the same quad, but facing in the other direction since we want this frond quad to be able to be viewed from both sides. We add the same four vertex positions, only this time in a counter-clockwise winding order. We also flip the sign of the vertex normals so that they are facing in the opposite direction.

```
CVertex( (vecPos + (vecRight * fX))                 , -vecOrtho, 1.0f, 1.0f ),
CVertex( (vecPos + (vecRight * fX)) + (vecAxis * fZ), -vecOrtho, 1.0f, 0.0f ),
CVertex( (vecPos - (vecRight * fX)) + (vecAxis * fZ), -vecOrtho, 0.0f, 0.0f ),
CVertex( (vecPos - (vecRight * fX))                 , -vecOrtho, 0.0f, 1.0f ),
```

We now have the horizontal quads built, so it is time to build the vertical ones so that we form a criss-cross shape of intersecting quads.

```
CVertex( (vecPos - (vecOrtho * fY))                 , -vecRight, 0.0f, 1.0f ),
CVertex( (vecPos - (vecOrtho * fY)) + (vecAxis * fZ), -vecRight, 0.0f, 0.0f ),
CVertex( (vecPos + (vecOrtho * fY)) + (vecAxis * fZ), -vecRight, 1.0f, 0.0f ),
CVertex( (vecPos + (vecOrtho * fY))                 , -vecRight, 1.0f, 1.0f ),
```

The vertex positions described above are shown in Figure 12.68. As you can see, this creates a quad aligned with the ortho vector instead of the right vector. As noted earlier, our ortho vector is actually pointing down and as such, the calculation of vertices V1 and V2 involve the subtraction of the ortho vector from the node position, while the calculations of vertices V3 and V4 involve its addition.

Notice that the vertex normal in each case we are passing is the negated right vector. The right vector is at right angles to the ortho vector and as such describes the plane of the quad we are building. The negated right vector will generate a vector pointing toward us in the diagram.

Finally, we want this quad to be two-sided so we add the same quad positions again, only this time with a reversed winding order and using the positive right vector as the vertex normals.

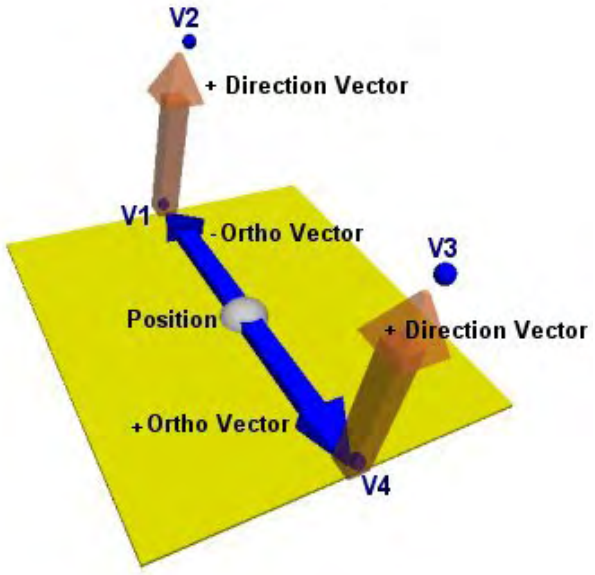


Figure 12.68

```
CVertex( (vecPos + (vecOrtho * fY))                  , vecRight, 1.0f, 1.0f ),
CVertex( (vecPos + (vecOrtho * fY)) + (vecAxis * fZ), vecRight, 1.0f, 0.0f ),
CVertex( (vecPos - (vecOrtho * fY)) + (vecAxis * fZ), vecRight, 0.0f, 0.0f ),
CVertex( (vecPos - (vecOrtho * fY))                  , vecRight, 0.0f, 1.0f )
};
```

We now have the 16 vertex positions stored in a CVertex array, so it is time to add them to the mesh. First we inform the passed CTriMesh of our intention to add 16 vertices so that it makes room at the end of its internal vertex array. We pass a pointer to the vertex array as well so that it can copy them.

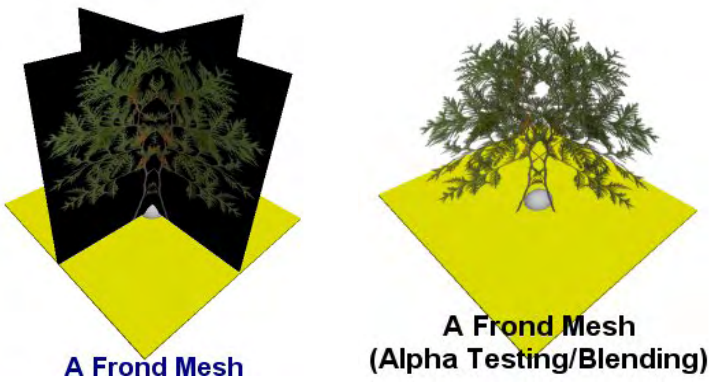


Figure 12.69

```
        // Add the frond vertex data
        long VIndex = pMesh->AddVertex( 16, pVertices );
        if ( VIndex < 0 ) return E_OUTOFMEMORY;
```

The AddVertex function returns the index (VIndex) of the first new vertex we added so that we know where we have to start indexing from when we build the indices.

For each quad we will need to add two triangles, each consisting of three indices (24 indices total). If we have a quad defined by vertices V1, V2, V3, and V4 in a clockwise winding order, then the first triangle will be created from V1, V2, and V3 and the second from vertices V1, V3, and V4. Below we see the code that allocates an array of 24 indices and indexes the vertices for each quad using this scheme. Of course, we must add VIndex to each zero based index value so that we index the positions in the mesh's vertex array where the frond vertices have been added. We cannot automatically assume they are the first 16 vertices in the mesh (since we know that they will not be because of prior branch vertex rings).

```
// Generate the index data for this frond
Index = (USHORT)VIndex;

USHORT pIndices[24] =
{ Index + 0 , Index + 1 , Index + 2 , Index + 0 , Index + 2 , Index + 3,
Index + 4 , Index + 5 , Index + 6 , Index + 4 , Index + 6 , Index + 7,
Index + 8 , Index + 9 , Index + 10, Index + 8 , Index + 10, Index + 11,
Index + 12, Index + 13, Index + 14, Index + 12, Index + 14, Index + 15 };
```

Finally, we inform the CTriMesh that we would like to add eight more triangles to its index buffer and pass the indices array so that it can copy them into its internal index array. Notice that in the third parameter to this function we pass in the number of the subset as 1 instead of 0. Thus, the branch segment faces will be assigned to subset 0 and the frond faces will be in subset 1. It makes perfect sense that we would need to put them in separate subsets since they both use different attributes (texture and material).

```
    // Add the frond face data (subset '1' )
    if ( pMesh->AddFace( 8, pIndices, 1 ) < 0 ) return E_OUTOFMEMORY;


    } // End if frond node

    // Success!!
    return D3D_OK;
}
```

This function will be called to add a frond whenever a frond node is encountered during the mesh building process. As you have seen, the code is actually quite easy to follow.

The last function that needs a minor update is the BuildSkinInfo function which is called from BuildNode when a branch mesh has been fully populated with its vertex and index data and is about to be turned into a skin. This function creates the ID3DXSkinInfo object that will be passed to the CreateMeshContainer function and contains the connection between bones in the hierarchy and the vertices they influence.

CTreeActor::BuildSkinInfo - Updated

This function has been modified in two places to support fronds, but the modifications are very small. It is suggested that you follow this discussion with the source code to Lab Project 12.2 handy so that you can see the changes in context. We will not be showing the entire function here since this was a big function and very little has changed.

You will recall that this function has a very simple job. It traverses the branch (it is only ever called for BRANCH_BEGIN nodes) stepping from node to node within the same branch and adding the vertex indices it collects between bone nodes to the skin info object. The first change happens near the start of the function, where we execute a while loop to step through the current node's child list to find the next node that is a continuation of the branch. Originally, we just ignored any other BRANCH_BEGIN nodes since we were only interested in finding the next node in the current branch we were processing. However, now the child list of a node can contain both BRANCH_BEGIN nodes and BRANCH_FROND nodes, so an extra check has been added to ensure that we also ignore frond nodes.

```
    while ( pSearchNode = pSearchNode->Child )
    {
        // We're not interested in other branches or fronds, so shift us through
        // until we find a node that is our segment, or end node.
        while ( pSearchNode &&
               (pSearchNode->Type == BRANCH_BEGIN ||
                pSearchNode->Type == BRANCH_FROND) )
                pSearchNode = pSearchNode->Sibling;

        // If we couldn't find a node, we have an invalid hierarchy
        if ( !pSearchNode ) return D3DERR_INVALIDCALL;

        // Was a frame created here?
        if ( pSearchNode->BoneNode ) BoneCount++;

    } // Next child segment
```

The final alteration is at the very bottom of the function's inner loop. You will recall from our earlier discussion that this function, once finding a bone node, will start collecting the indices of all child nodes until the point that another bone is encountered. At this point all the indices currently collected in the temporary indices array are added for the previous bone in the ID3DXSkinInfo object. A new section has now been added which essentially just says, if we are processing a frond node, add the frond vertex indices to the temporary indices array also. If you have the source code in front of you, this addition will seem perfectly logical.

```
            else if ( pSearchNode->Type == BRANCH_FROND )
            {
                // Add influences for each of the 16 frond vertices
                for ( i = 0; i < 16; ++i )
                {
                    pIndices[ InfluenceCount ] = IndexCounter++;
                    pWeights[ InfluenceCount ] = 1.0f;
                    InfluenceCount++;
```

```
                } // Next index

            } // End if frond

        } // Next node sibling

        // If we couldn't find a segment node, we've reached the end of the branch
        if ( !pSegmentNode ) break;

    } // Next child segment

    // Clean up
    delete []pIndices;
    delete []pWeights;

    // Success!!
    return D3D_OK;
}
```

We have now covered all the functions that need to be altered in the tree generation process and our CTreeActor class is now capable of creating trees with fronds. This certainly acts to increase our visuals by quite a significant margin.

However, we are not quite done yet in terms of our overall support for fronds. In the previous lab project, we could just allow the base class (CActor) DrawActor and DrawActorSubset methods to be called to render the tree, but this is no longer the case. We will need to override these functions so that when rendering frond subsets we can enable alpha blending and alpha testing before calling the base class versions of the function. Let us have a look at the CTreeActor::DrawActorSubset function first.

CTreeActor::DrawActorSubset

This function is extremely simple since it calls the base class implementation to do the actual rendering. All it does is test to see if the passed attribute ID matches the attribute ID of the frond subset (which we stored earlier in the m_nFrondAttribute member variable). If so, it enables alpha testing and alpha blending.

```
void CTreeActor::DrawActorSubset( ULONG AttributeID )
{
    // Is this our frond attribute?
    if ( m_Properties.Include_Fronds && AttributeID == m_nFrondAttribute )
    {
        // Setup states
        m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, TRUE );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHAREF, (DWORD)0x000000A0 );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
        m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
        m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
        m_pD3DDevice->SetTextureStageState( 0,D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
```

```
        m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );

    } // End if frond attribute

    // Call underlying draw
    CActor::DrawActorSubset( AttributeID );

    // Reset states if applicable
    if ( m_Properties.Include_Fronds && AttributeID == m_nFrondAttribute )
    {
        m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, FALSE );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );

    } // End if frond attribute
}
```

We first enable alpha testing because alpha blending by itself will not suffice. We do not want the black background pixels to be rendered at all. Although these pixels have zero alpha components and would not alter the contents of the frame buffer when rendered, with alpha blending enabled, they would actually still be rendered. Although these pixels would be blended with the frame buffer with zero weight (leaving the frame buffer unchanged) their depth values would still be written to the depth buffer. Therefore, the black space around the leaves would block leaves located behind from being drawn as they would be rejected by the depth test. In essence, it would seem as if the leaves were blocked by invisible geometry.

Figure 12.70 : Low Alpha Testing Reference Value (Ref = 5)

By enabling alpha testing we can make sure that these background pixels get rejected in the pixel pipeline. This means their depth values will not be recorded in the depth buffer and therefore, we will not have these issues. We set the alpha testing reference value to a rather arbitrary 0xA0 (160) so that all pixels with an alpha value of less than 160 get rejected. The reason we do not set this to a value such as 1 for example (so only totally transparent pixels such as the background pixels get rejected) is because, when we generate an alpha channel, the pixels between the opaque leaf edges and the transparent black background will often also be blended from opaque to transparent. Therefore, this makes sure that we remove nearly all the pixels that are mostly transparent, but leave the ones in place that are partially transparent since these pixels may contain part of the leaf edge. Setting the alpha reference to a low value (e.g., 1) will remove all the background pixels but will leave the partially transparent pixels around the opaque leaves in place and their depth values will still be written to the frame buffer. This provides an unattractive border around the leaves (see Figure 12.70). As you can see, there is a large number of pixels that are not totally transparent that do not get rejected by the alpha testing pipeline, so setting a higher reference value helps filter these undesirables out.

Figure 12.71 Alpha Blending Disabled

We then enable alpha blending so that the leaves get blended with the frame buffer and any partially transparent pixels around the leaf edges which did not get rejected from the alpha test will allow for objects behind them to show through. If you play around with the alpha testing reference value you will see exactly why this is necessary. If we do not enable alpha blending, the results will not look overly bad, but there will still be a hard edge to leaves (see Figure 12.71). Enabling alpha blending allows us to keep the outer portions of the leaves that are partially transparent but have them blend out smoothly, removing the hard edge around the leaves. You can experiment with the settings to find what works best for you. Obviously, while alpha blending does seem to produce better results, it adds some additional cost.

CTreeActor::DrawActor

The CActor::DrawActor function can be used to automate the rendering of managed mode meshes. It is called once by the application and will render all of its subsets. It can do this because a managed mode mesh contains its own texture and material arrays and can set them before rendering each of its subsets.

We also have to override this function in CTreeActor for the same reason as before (to enable the alpha testing and alpha blending render states). Unfortunately, when rendering a managed mode tree we no longer have per-subset control to decide whether we should enable the states for the entire rendering of the tree or not. This means that if the tree does have fronds, we will have to enable alpha testing and alpha blending and then use the base class implementation to instruct the mesh to draw all its subsets. Sadly, this does mean that both subsets will be rendered with these states enabled. These states will not harm the rendering of the non-frond subset, but it is not ideal to have these states set when we do not wish to use them since they introduce overhead in the pixel pipeline.

```
void CTreeActor::DrawActor( )
{
    // Fronds included?
    if ( m_Properties.Include_Fronds )
    {
        // Setup states
        m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, TRUE );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHAREF, (DWORD)0x000000A0 );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
        m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
        m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
        m_pD3DDevice->SetTextureStageState( 0,D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
        m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );

    } // End if fronds included

    // Call underlying draw
    CActor::DrawActor();
```

```
    // Reset states
    if ( m_Properties.Include_Fronds )
    {
        m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, FALSE );
        m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );

    } // End if fronds included
}
```

Conclusion

This concludes our discussion of trees (for now) and also brings us to the end of our skinning discussions in this course. There was quite a lot of challenging material to take in at times, but you should now have a very strong understanding of frame hierarchies and skinned meshes. We now have an actor that fully supports skinning which we can use in all of our applications from this point on. Additionally we have a pretty nice derived actor class that allows us to generate trees for our various scenes. We recognize that there was a lot to digest in this chapter, so take your time and re-read the sections that you found difficult before moving on to the workbook.

Workbook Twelve:
Skinning Part II

Introduction

This workbook marks the halfway point in this course. As such, one of the things we will endeavor to do is bring together many of the components we have developed throughout the course into a single application. In the accompanying textbook we learned how to generate trees. Trees will be just one element of what we intend to incorporate in this lesson.

In previous lessons our applications have essentially taken one of two forms; either a terrain demo or a scene demo. Now we will begin the process of merging these two concepts. We will add support to our IWF loading code for terrain entities so that a single IWF file can contain multiple pieces of terrain and interior scenes and objects. Previously, our applications have used a single terrain and as such the vertices in each sub-mesh of the terrain were defined in world space. Because we now wish to support multiple terrains that may be positioned at different locations throughout the scene, we will have to modify our terrain class (only slightly) so that it takes into account a world matrix for that terrain during rendering. The CScene class will now have to be able to store an array of CTerrain pointers and render them all instead of assuming only a single terrain exists.

We will also add loading support for tree entities so that objects of type CTreeActor will be added to the scene for each tree entity found in the IWF file. Internally, GILES™ uses the CTreeActor code that we developed in the accompanying textbook for its tree support. This means the creation properties stored inside the IWF file for a tree entity will work perfectly with our CTreeActor class and generate the same tree formation as displayed in GILES™. The loading code can simply instantiate a new CTreeActor for each entity it finds and feed in the tree creation parameters pulled straight from the IWF file.

Other entity types will also have support added to the IWF loading code. The GILES™ fog entity is another example. This is a simple entity that stores information about how the Direct3D fog parameters should be set during rendering.

By the end of this workbook our framework will be in a much nicer state to work with. We will be able to feed it IWF files that contain internal geometry, external reference entities for animated X files, multiple terrains, trees, and fog. Our framework will know how to store and render these components as a single scene.

While all of these housekeeping tasks and upgrades are important, they will not be the only ideas introduced. One of the main topics we will explore in this workbook will be in the development of a data driven animation system. This system will allow our applications to manage the setup and playback of complex animation sequences through a very user friendly interface. It will allow our applications (or any subsequent component) to communicate a command such as “Walk and Shoot’ to an animated actor, and the actor will know which animation sets should be set and activated, which ones should be deactivated, and which ones should be blended from one to another. Although this sounds like a complex system to implement, it will expose itself to the application via only two CActor methods. This will all be demonstrated in Lab Project 12.3 where we will see an animated skinned character walking around a terrain. The character will perform several animations such as, “Walk at ease”, “Walk and Shoot” and “Idle” when ordered by the application. Our “Action Animation System” (discussed later)

will be managing which animations sets should be assigned to which tracks on the animation mixer and which animation sets are currently in the process of having their contributions faded in or out.

This also means that we will finally have something nice to look at when we place our camera system into third person mode. When we developed our camera system way back in Chapter Four of Module I we were not yet equipped to add a real animated character to our application. As a result, our third person camera was forced to use a simple cube as a placeholder for where the character should be. The need for that placeholder is now gone and we are ready to attach an animated skinned character to the CPlayer object instead. The scene class will have a new function added that will be called by the application to load a character into a CActor object which will be attached to the CPlayer object. It will load the X file of the character mesh and its animation data and register any callback functions with the CPlayer object that are necessary for this loading process. This function will also load an .ACT file that will be used to populate the actor with information about the various actions the character can perform.

We will discover shortly that .ACT files are simple .ini files that we can create to feed information into our duly named Action Animation System (AAS for short). An .ACT file will contain one or more *actions*. Each action has a name (e.g., "Walk at Ease") and contains the names and properties of a number of animation sets that need to be set on the controller in order for that action to be played. An action called "Observe" for example might consist of multiple animations sets. It may wish to play an animation set that works on the legs to place them into a crouch position, another animation set for the torso that raises the character's binocular holding hands up to his face, and perhaps another animation set that tilts the head of the character to look through the eye pieces. An action is really just a collection of animation set names along with properties like their speed and weight. These actions will be loaded and stored by the actor and used by its AAS to set up the controller in response to the application requesting a certain action. The action system will take care of determining which tracks are free for a given animation set to use and how to configure that track to play the requested action correctly. All the application has do to is call the CActor::ApplyAction method and specify the name of the action it would like to play that matches the name we assign it in the .ACT file.

Goals for this lesson:

- Implement the Action Animation System to act as a middle tier between the application and the actor's underlying controller. The AAS will allow the application to launch complex animations in response to game events or user input with the minimum of work. A simple function call is all that will be needed to perform complex blending between groups multiple animation sets.
- Add a function to load and populate a CActor for use as the CPlayer's third person object.
- Unify components such as heightmap based terrains, trees, and interior scenes into a single application.
- Upgrade our loading code to support loading terrains definitions, tree definitions, and fog parameters directly from the IWF file. These definitions can be used as the construction parameters for our CTerrain and CTreeActor classes.
- Upgrade CTerrain to work in model space and use a world matrix for rendering and querying.

Lab Projects 12.1 and 12.2: Trees

There is very little to discuss for the first two lab projects since the discussion of our tree class was pretty much completed in the textbook. Lab Projects 12.1 and 12.2 simply load a GILES™ tree entity and then use our CTreeActor class to build the tree based on the information stored in the IWF file. It is essentially just our mesh viewer with an updated ProcessEntities function that supports the loading of GILES™ tree entities.

The GILES™ tree entity has an identifier of 0x205, so we will create a #define for this value for entity ID comparisons during the loading process. The entity does not contain any geometry; it is just a collection of tree growth properties which we can feed into our CTreeActor::GenerateTree method.

Excerpt from CScene.h

```
#define CUSTOM_ENTITY_TREE      0x205   // Tree entity identifier
```

The GILES™ tree entity plug-in uses the CTreeActor code we developed in this lab project for its tree creation and rendering, so the properties stored in the IWF file are compatible with CTreeActor.

The tree entity data also has a flags member which describes how the tree should be generated. The flags are shown below. Once again we have created #define's for these in CScene.h to use during the loading process.

Excerpt from CScene.h

```
#define TREEFLAGS_INCLUDEFRONDS     0x1 // Tree actor to include fronds
#define TREEFLAGS_GENERATEANIMATION 0x2 // Tree should generate animation on load
#define TREEFLAGS_INCLUDELEAVES     0x4 // Tree should generate leaf information
```

The data area of the GILES™ tree entity is laid out in a very specific way, so we will set up a structure that mirrors its layout so that we can easily extract the values from the file into this structure. The TreeEntity structure is defined in CScene.h. The Flags member can be set to a combination of the flags listed above. The TREEFLAGS_INCLUDELEAVES flag is currently not supported by our lab project. It will be used in Module III when we revisit the subject of tree rendering.

```
typedef struct _TreeEntity
{
    ULONG       Flags;              // Tree actor flags
    char        BarkTexture[1024];  // The texture to use for the bark of the tree
    char        FrondTexture[1024]; // The texture to use for the frond geometry

    ULONG       GrowthSeed; // The random seed used to build the tree
    TreeGrowthProperties    GrowthProperties;    // The tree growth properties.
    D3DXVECTOR3             AnimationDirection;  // Animation direction vector
    float                   AnimationStrength;   // Animation 'strength'

    D3DXVECTOR3             InitialDimensions;   // Initial branch dimensions
    D3DXVECTOR3             InitialDirection;    // Initial growth direction.

} TreeEntity;
```

The members of this structure should be obvious to you. The Flags member contains whether or not the tree should have fronds created and/or animation data created for it. The next two members contain the names of the image files to use for the bark and frond textures. The Growth seed is the value that the random number generator should be set to during tree creation. As we have seen, we can pass this straight into the CTreeActor::GenerateTree function. The next member is the TreeGrowthProperties structure discussed in the textbook. Following that are the wind direction and wind strength used for generating animation. Finally we have the initial dimensions and growth direction for the root branch node.

CScene::ProcessEntities - Updated

The CScene::ProcessEntities function is very familiar by now since it has been part of our loading framework for quite some time. It is called by the CScene::LoadSceneFromIWF function to extract any entity information our scene may be interested in supporting. It is passed a CFileIWF object, which at this point has loaded all the entity data and stored it in its m_vpEntityList STL vector. This function loops through the vector and extracts any information it wishes to use. Until now, this function has only supported the loading of light entities, reference entities, and sky box entities. We will now add support for GILES™ tree entities.

Because this function is rather large, we will snip out all the code that processes the entities we have already covered and replace this code with "……SNIP……".

```
bool CScene::ProcessEntities( const CFileIWF& File )
{
    ULONG      i, j;
    D3DLIGHT9  Light;
    USHORT     StringLength;
    UCHAR      Count;
    bool       SkyBoxBuilt = false;

    // Loop through and build our lights & references
    for ( i = 0; i < File.m_vpEntityList.size(); i++ )
    {
        // Retrieve pointer to file entity
        iwfEntity * pFileEntity = File.m_vpEntityList[i];

        // Skip if there is no data
        if ( pFileEntity->DataSize == 0 ) continue;
```

The first section of code (shown above), sets up a loop to step through every entity in the CFileIWF object's internal entity vector. Inside the loop we get a pointer to the current entity to be processed (as an iwfEntity structure). We then read the entity's data size and if we find it is zero, we skip it. An entity with a data size of zero is probably one that has been erroneously written to the file.

In the next section of code we test to see if the current entity is a light entity . If it is and we have not already loaded the maximum number of lights, we add a light to the scene. The code that adds the light to the scene's light array has been snipped since this code has been with us for a very long time.

```
        //skip all lights if we've already reached our limit, or maximum
        if ( (m_nLightCount < m_nLightLimit && m_nLightCount < MAX_LIGHTS)
              && pFileEntity->EntityTypeMatches( ENTITY_LIGHT ) )
        {
            …… SNIP ……

        } // End if light
```

The next code block is executed if the current entity has an author ID of ‘GILES’. If it does then we know this is either a sky box entity, a reference entity, or a tree entity; all of which we wish to support in our loading code. When this is the case we set up some entity structures that will be used to extract the information into. Notice that the new addition here is the instantiation of a TreeEntity structure. We then get a pointer to the entity’s data area.

```
        else if ( pFileEntity->EntityAuthorMatches( 5, AuthorID ) )
        {
            SkyBoxEntity SkyBox;
            ZeroMemory( &SkyBox, sizeof(SkyBoxEntity) );

            ReferenceEntity Reference;
            ZeroMemory( &Reference, sizeof(ReferenceEntity ));

            TreeEntity Tree;
            ZeroMemory( &Tree, sizeof(TreeEntity));

            // Retrieve data area
            UCHAR * pEntityData = pFileEntity->DataArea;
```

Now that we have a pointer to the entity data area we will test the entity ID to see what type of GILES™ entity it is. We set up a switch statement to handle skybox, reference, and tree entity types. As we have already covered the code for the first two in previous lab projects, the code has been snipped here.

```
            switch ( pFileEntity->EntityTypeID )
            {
                case CUSTOM_ENTITY_REFERENCE:

                    …… SNIP ……

                    break;

                case CUSTOM_ENTITY_SKYBOX:

                    …… SNIP ……

                    break;
```

In the next section of code we see the case that is executed when the entity type is a GILES™ tree entity. Extracting the values from the data area is simply a case of copying the correct number of bytes from the entity data pointer (retrieved above) into our TreeEntity structure for each property, and then advancing this pointer so that it points at the next value to be read. In the section shown below we

extract the flags ULONG and the string containing the bark texture filename. Notice how we are copying each value into their corresponding members in the TreeEntity structure.

```
                case CUSTOM_ENTITY_TREE:

                    // Retrieve the fog entity details.
                    memcpy( &Tree.Flags, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    // Bark Texture
                    memcpy( &StringLength, pEntityData, sizeof(USHORT) );
                    pEntityData += sizeof(USHORT);

                    if ( StringLength > 0 )
                    memcpy( Tree.BarkTexture, pEntityData, StringLength );
                    pEntityData += StringLength;
```

The next value in the entity data area of the tree is a value describing how many frond textures our tree uses. We currently only support a single frond texture so we skip all but the first texture listed (feel free to experiment with this setting as you please). We then extract the first texture name into the TreeEntity structure.

```
                    // Frond Texture Count
                    Count = *pEntityData++;

                    // Loop through textures in the file
                    for ( j = 0; j < (ULONG)Count; ++j )
                    {
                        // Get string length
                        memcpy( &StringLength, pEntityData, sizeof(USHORT) );
                        pEntityData += sizeof(USHORT);

                        // skip all others stored.
                        if ( j < 1 )
                        {
                            // Read String
                            if ( StringLength > 0 )
                            memcpy( Tree.FrondTexture, pEntityData,StringLength );
                            pEntityData += StringLength;

                        } // End if first frond texture
                        else
                        {
                            // Skip String
                            pEntityData += StringLength;

                        } // End if unsupported extras

                    } // Next Frond Texture
```

Notice that in the frond texture name loop, we skip all but the first texture found. For all others we still increment the data pointer to move it past any other frond texture names that may be stored there. At the

end of this loop, regardless how many frond texture names were stored, our data pointer will be pointing at the next property to be extracted.

The GILES™ tree entity also supports a concept of separate leaves (in addition to the fronds), but our class does not currently support this technique (Module III will address this). Therefore, we will simply retrieve the number of leaves stored in the file and will then increment the data pointer past the string stored for each leaf.

```
                    // Leaf Texture Count
                    Count = *pEntityData++;

                    // Loop through leaf textures in the file
                    for ( j = 0; j < (ULONG)Count; ++j )
                    {
                        // Get string length
                        memcpy( &StringLength, pEntityData, sizeof(USHORT) );
                        pEntityData += sizeof(USHORT);

                        // Leaves are not yet supported, skip all texture names.
                        pEntityData += StringLength;

                    } // Next Leaf Texture
```

Next we extract the random generator seed value of the tree into the tree entity structure.

```
// Growth Seed
memcpy( &Tree.GrowthSeed, pEntityData, sizeof(ULONG) );
pEntityData += sizeof(ULONG);
```

The next block of data stored in the IWF file is basically a mirror of the TreeGrowthProperties structure. While the following block of code looks rather complex, all it is doing is fetching the values for each property in order and copying them into the TreeGrowthProperties member of the TreeEntity structure.

```
// Growth Properties
memcpy( &Tree.GrowthProperties.Max_Iteration_Count,pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Initial_Branch_Count,pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Min_Split_Iteration,pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Max_Split_Iteration,pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Min_Split_Size, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Max_Split_Size, pEntityData, sizeof(float) );
pEntityData += sizeof(float);
```

```
memcpy( &Tree.GrowthProperties.Min_Frond_Create_Iteration,
         pEntityData, sizeof(USHORT) );

pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Frond_Create_Chance, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Frond_Min_Size, pEntityData, sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);

memcpy( &Tree.GrowthProperties.Frond_Max_Size, pEntityData, sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);

memcpy( &Tree.GrowthProperties.Two_Split_Chance, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Three_Split_Chance, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Four_Split_Chance, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Split_End_Chance, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Segment_Deviation_Chance,
        pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Segment_Deviation_Min_Cone,
         pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Segment_Deviation_Max_Cone,
        pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Segment_Deviation_Rotate,
         pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Length_Falloff_Scale, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Split_Deviation_Min_Cone,
         pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Split_Deviation_Max_Cone,
        pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy(&Tree.GrowthProperties.Split_Deviation_Rotate, pEntityData,sizeof(float) );
pEntityData += sizeof(float);
```

```
memcpy( &Tree.GrowthProperties.SegDev_Parent_Weight, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy(&Tree.GrowthProperties.SegDev_GrowthDir_Weight,pEntityData,sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Branch_Resolution, pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Bone_Resolution, pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

memcpy( &Tree.GrowthProperties.Texture_Scale_U, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Texture_Scale_V, pEntityData, sizeof(float) );
pEntityData += sizeof(float);

memcpy( &Tree.GrowthProperties.Growth_Dir, pEntityData, sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);
```

After the TreeGrowthProperties structure we have the animation wind direction vector and the wind strength, so let us read these values next (a 3D vector and a float)

```
// Animation Properties
memcpy( &Tree.AnimationDirection, pEntityData, sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);

memcpy( &Tree.AnimationStrength, pEntityData, sizeof(float) );
pEntityData += sizeof(float);
```

Finally, we read in the 3D vector describing the initial dimensions of the root node of the tree and its growth direction (also a 3D vector).

```
// 'Initial' values
memcpy( &Tree.InitialDimensions, pEntityData, sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);

memcpy( &Tree.InitialDirection, pEntityData, sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);
```

We have now loaded all the data from the entity into our TreeEntity structure. Next we test to see if the flags we extracted have the TREEFLAGS_INCLUDEFRONDS bit set. If so, we set the Include_Fronds Boolean of the TreeGrowthProperties structure to true so that when we feed it into the CTreeActor::GenerateTree function, it will know to generate fronds for the tree.

```
// Setup data from flags
if ( Tree.Flags & TREEFLAGS_INCLUDEFRONDS )
     Tree.GrowthProperties.Include_Fronds = true;
```

At this point the TreeGrowthProperties structure stores all the tree information we need, so let us generate the tree. This task is actually handed off to the CScene::ProcessTree method. This is a new method that we have added to this class and will discuss next.

```
// Process the tree entity (it requires validation etc)
if ( !ProcessTree( Tree, (D3DXMATRIX&)pFileEntity->ObjectMatrix ) ) return false;

break;

} // End Entity Type Switch
```

The ProcessTree method takes two parameters. For the first parameter we pass the TreeEntity structure that we have just populated with the data from the entity and as the second parameter we pass in the entity's world matrix (every entity has a world matrix member describing its location in the scene).

```
        } // End if custom entities

    } // Next Entity

    // Success!
    return true;
}
```

With the changes to the ProcessEntities function covered let us now look at the new CScene::ProcessTree function. This function has the task of creating a new CTreeActor using the values stored in the passed TreeEntity structure and adding it to the scene's CObject array,

CScene::ProcessTree

The first section of this function (shown below) creates a new CTreeActor and registers a scene attribute callback function with it. It then sets the tree actor's branch and frond material to the filenames stored in the passed TreeEntity structure and calls the CTreeActor::GenerateTree method to build the tree meshes.

```
bool CScene::ProcessTree( const TreeEntity& Tree, const D3DXMATRIX & mtxWorld )
{
    HRESULT       hRet;
    CTreeActor  * pTreeActor  = NULL;
    CObject     * pTreeObject = NULL;

    // Allocate a new tree actor
    pTreeActor = new CTreeActor;
    if (!pTreeActor) return false;

    // Setup and generate the tree
    pTreeActor->RegisterCallback( CActor::CALLBACK_ATTRIBUTEID,
                                  CollectAttributeID, this );

    pTreeActor->SetBranchMaterial( Tree.BarkTexture, NULL );
```

```
    pTreeActor->SetFrondMaterial( Tree.FrondTexture, NULL );

    pTreeActor->SetGrowthProperties( Tree.GrowthProperties );

    hRet = pTreeActor->GenerateTree( D3DXMESH_MANAGED,
                                     m_pD3DDevice,
                                     Tree.InitialDimensions,
                                     Tree.InitialDirection,
                                     Tree.GrowthSeed );

    if ( FAILED(hRet) ) { delete pTreeActor; return false; }
```

At this point the tree skins have been built so we will check the flags member of the passed TreeEntity structure to see if animation should be generated for this tree. It so, we pass the wind direction and the wind strength properties into the CTreeActor::GenerateAnimation method.

```
    // Generate the animation if requested
    if ( Tree.Flags & TREEFLAGS_GENERATEANIMATION )
    {
        hRet = pTreeActor->GenerateAnimation( Tree.AnimationDirection,
                                              Tree.AnimationStrength );
        if ( FAILED(hRet) ) { delete pTreeActor; return false; }

    } // End if generate animation
```

With our CTreeActor now completely built, let us make room at the end of the scene's CActor array to store its pointer.

```
    // Store this new Actor
    if ( AddActor( ) < 0 ) { delete pTreeActor; return false; }
    m_pActor[ m_nActorCount - 1 ] = pTreeActor;
```

With the tree actor's pointer added to the scene's actor array, we store the actor and its world matrix in a newly allocated CObject before resizing the scenes CObject array and storing CObject's pointer in it.

```
    // Now build an object for this TreeActor (standard identity)
    pTreeObject = new CObject( pTreeActor );
    if ( !pTreeObject ) return false;

    // Copy over the specified matrix
    pTreeObject->m_mtxWorld = mtxWorld;

    // Store this object
    if ( AddObject() < 0 ) { delete pTreeObject; return false; }
    m_pObject[ m_nObjectCount - 1 ] = pTreeObject;

    // Success!!
    return true;
}
```

That completes the framework changes for Lab Project 12.2. Our framework now has support for loading GILES™ tree entities and generating skinned actors for them.

Lab Project 12.3

Lab Project 12.3 is the first project that will employ our Action Animation System. It will be implemented into the CActor interface and used by the application in this lab project to apply animated sequences to the character as he navigates the level. In this project we will also update our framework and loading code to add IWF file terrain and fog support. The first order of discussion in this section will be the design and implementation of the Action Animation System.

The Action Animation System

The goal of the action system we will develop is not to provide the most flexible animation mixing system imaginable. Indeed such a system would be quite complex and hard to use as a teaching exercise. Instead, our goal is to provide a simple way to describe how the various animation controller tracks should be set up for a specific action, while providing a relatively easy to use interface for the end user. Let us be clear about what we mean by an action.

What is an Action?

In our system, we assign the word ‘action’ to the collection of animation sets, and their associated properties, that must be set on the animation controller for a high level action to be played. It is quite common for an artist to create animation sets for a given model such that they are localized to various regions of the body. As an example, a character may have three animation sets created for it that animate only the legs of the character in a certain way. One animation set might be called ‘Legs Idle’ and simply shuffles the legs from side to side while the character is awaiting further instruction. A second animation set might be called ‘Legs Walk’ and needs to be played when the character is walking. A third animation set might be called ‘Legs Run’ and when played, launches the legs of the character into a full sprint. Remember, these animation sets affect only the legs. Of course, the artist would also need to define animation sets for the torso. For example, there may be two animation sets called ‘Torso Ready’ and ‘Torso Shooting’. When the ‘Torso Ready’ animation set is played, the character should hold his weapon in his hands ready for action but not yet taking an aggressive posture. We might think of this as being like the arms of a soldier as he walks on patrol with his assault rifle. The animation set ‘Torso Shooting’ could really be as simple as a single keyframe set that when played, poses the arms of the soldier up near his chest as if the soldier is firing a weapon. Of course, animation sets may exist for other parts of the body too; there may be animation sets that tilt the head of the soldier to one side when in the firing position such that he is looking through the eye piece of his weapon. For now, we will just go forward with the arms and legs example for ease of explanation.

Although the artist could create animation sets that animate both the torso and the legs, the benefits of the segmented approach is that it relieves the artist from having to create animation sets for every combination of torso and leg animations that the application may wish to play (although to be sure, many game development shops will do exactly that). Imagine that the artist created a single animation set called ‘Walk and Shoot’ that animated both the legs in a walking motion and the arms in a shooting

motion. The artist would also have to create such animation sets for every combination. For example, he may also have to create the following animation sets “Run and Shoot”, “Idle and Shoot”, Walk and Ready”, “Run and Ready”, “Idle and Ready”. When you imagine the various animation sequences which can be applied to the legs alone “Jump, Crouch, and Kick”, the artist would then have to create ‘Ready’ and ‘Shoot’ versions of these animations too. As you might assume, this would put a heavy asset creation burden on the artist.

By using a segmented approach as discussed above, the artist can create isolated animations for the various parts of the character’s body. These separate components can then be mixed and matched by our engine at run time. The artist would create only three animation sets for the legs in this example. ‘Legs Idle’, ‘Legs Walk’, and ‘Legs Run’. If we imagine that the character is running and then wants to fire his weapon, we can still leave the ‘Legs Run’ animation set playing but then also play the ‘Shooting’ animation set. Since these two sets, which are playing simultaneously, work on totally separate parts of the body, they do not conflict with each other. If the character is still firing his weapon but then would like to start walking instead, we can just swap out the ‘Legs Run’ animation for the ‘Legs Walk’ animation. The legs would now play a different sequence while the torso continues to fire the weapon, unhindered by the change applied to the legs.

Although the segmented animation set design is often desirable, it does shift the burden to the application with respect to handling which animation sets should be played simultaneously to perform a given action. For example, we know that in the previous example, if we wanted to have our character perform an action such as ‘Run and Fire’, we would have to assign both the ‘Legs Run’ and ‘Shooting’ animation sets to active tracks on the animation mixer. This is one of the core ideas that our Action Animation System addresses. The important thing at the moment is for us to understand that an action can be thought of quite abstractly as an action to be performed by the object. For instance, we may want our actor to swim, jump, run, and walk. But each of these actions may need to be comprised of many different animation sets under different circumstances. An action (in the terms outlined by this system) is that collection of animation sets (and their properties) which describes to the animation controller which components are required to carry out the act of, for instance, swimming or walking.

By integrating this system directly into our Actor class, it allows us to use a simple data driven approach for setting up the controller, without having to necessarily work through separate components. This keeps the complexity of the system low from a user standpoint. Because are not yet ready to employ a full scripting system (we will introduce you to that in Module III), we currently make use of a static descriptor file format. Because of the static nature of the description components, we will be bound to a ‘pull only’ process (i.e. it cannot change based on the circumstances at any given time during game-play). These descriptor files are essentially just standard Windows™ ‘*.ini*’ files, which we process by making use of the standard Win32 API functions `::GetPrivateProfileInt()`, and `::GetPrivateProfileString()`. We will discuss the format of these files in a moment and examine their contents. An Action file will have an ‘.act’ extension and contain some number of actions. Each action will have a name that the application will use to apply that action to the character (e.g., “Walk at Ease” or “Run and Shoot”). Each action in the .act file will contain a list of the names of animation sets which need to be played simultaneously to achieve that action as well as the properties that should be assigned to the tracks for those animation sets. This means we can also control the speed and contribution strength of a given animation set within an action when it is applied.

Although the ability to store a list of animation sets and their properties with an assigned action name would be a convenient way for the application to specify the various animation sets that should be set on the controller to achieve a given action, our action system provides a powerful blending feature that will allow it to intelligently make decisions about which tracks on the mixer should be used. It will also determine which animation sets from a previously played action should be slowly diminished as we phase in an animation set from a newly applied action. The ability for us to be able to blend the animation sets from a previous action into the animation sets of a new action over time is an essential component of our system.

Action Blending

To understand why blending is useful (and often necessary) let us continue with our previous example of the two animation sets defined for the torso of the character; 'Ready' and 'Shooting'. We will also assume for the sake of this discussion that both these animation sets contain just a single keyframe. That is, they do not really animate the torso; they just contain a snapshot of the character's arms in a different position. When the 'Ready' animation set is playing the arm frame matrices are set such that the soldier is holding his weapon in front of his chest. The 'Shooting' animation set is similarly a single keyframe looping set that when played, positions the arms of the character such that he is holding his weapon up to his face.

Figure 12.2 shows the state of the torso when each animation set is being played. Let us now imagine that the character's torso is currently in the 'Ready' pose and we wish it to assume the 'Shooting' pose when the user presses the fire button. The wrong way to handle this, when the fire button is pressed, would be to remove the 'Ready' pose from its track on the animation mixer and assign the 'Shooting' pose in its place. The next time the scene is rendered, the arms of the character will be in the shooting pose as requested, but this is a terrible way to accomplish that objective.

Figure 12.2

The obvious problem with this approach of course is the lack of any smooth transition between states. Simply swapping one pose for another would mean that the arms of the character would abruptly change from the ready position to the shooting position within the time of a single frame update.

In reality, we would expect to see the character's arms slowly transition from the ready pose to the shooting pose. But if we do not employ any blending support at the code level, the artist would have to make transition animations. When we consider the sheer number of animations an artist would need to make for each pose for each section of the character's body, and then consider that the artist would then need to build transition animation sets from each pose to every other pose, the asset demands on the

artist for just a single character would be staggering. By exposing run-time blending support in our action system, we can get those transitions for free. So let us now discuss how our system may provide asset-free transition sequences.

When we think about the problem in Figure 12.2 and factor in the functionality that is exposed by the D3DX animation system, a solution arises that makes use of sequencer events and the weight properties of mixer tracks.

Each action in the .act file will contain a number of animation set definitions. A *set definition* is comprised of the name of one of the animation sets defined in the X file along with properties that will be applied to the mixer track when the animation set is played. One property that we will also introduce is the ability to assign an animation set a 'Group' name. Within a given action, there should never be more than one animation set with the same group name, but animation sets contained in different actions can belong to the same group. For example, we might have two actions defined in our .act file. In each action there might be an animation set definition that describes how the torso of the character should be animated. (It is important to note that only one action can be applied to the character at any one time.) We will imagine that the animation set in the first action that animates the torso is called 'Ready' and the torso animation set being used by the second action is called 'Shooting'. Because both of these animation sets exist in different actions, but animate the same region of the character, we will assign them the same group name (e.g., 'Arms').

Now let us imagine that our application applies the first action. The 'Ready' animation set will be assigned to the first empty track on the mixer to animate the arms of the character in the ready pose. We will cache some information in the actor at this point that says that the animation set currently assigned to that track belonged to a group with the name 'Arms'. So far, nothing out of the ordinary is happening until the user presses the fire button and the second action is applied. We know that the second action contains as one of its animation sets the 'Shooting' animation set, which also belongs to a group with the name 'Arms'. Here is where things get interesting. Rather than just replacing all the animation sets that were set on the controller when the previous action was applied with the animation sets defined in this new action, we will do the following for any set that has been configured for blending in the action file:

- For each animation set definition in the action we are about to apply we will search for a track on the controller that is not in use (i.e., a free track).
- If the animation set has been configured to blend in the .act file, it will also have a property assigned to it called Mix Length. This is a value describing the number of seconds we would like the animation set's weight to rise from zero to full strength. For example, a value of 5 seconds means that we would like this animation set to slowly fade in to full contribution over five seconds.
- At this point we register a sequencer event with the new track. The event is a track weight event that describes our fade in from zero to full strength with a duration of five seconds, starting immediately.
- Since only one animation set from a given group should be current at one time, we deactivate any animation sets that belong to the same group as the one we have just activated. In our example, we have just set the 'Shooting' animation set on a mixer track. Since this set belongs to the 'Arms' group, we should now deactivate any animation sets assigned by a previous action

that belongs to the same group. In our example, the ‘Ready’ animation set would currently be in use by the mixer and is part of the ‘Arms’ group.

- Since the ‘Shooting’ set was configured to blend in over a period of five seconds, we will do the opposite for any animation set that belongs to the same group that was applied by a previous action. That is, we will set a sequencer event that will fade the weight/contribution of the ‘Ready’ set from its current full strength setting to zero over a period of five seconds.
- We will also set a second sequencer event that will be triggered in 5 seconds time (when the set has fully faded out) to disable the track with the ‘Ready’ set.
- We will cache the information somewhere that this track is now fading out so that it will not be used by any future actions we apply until the fade out is complete.
- When any tracks have their enabled state set to false by the sequencer event, we know that the set that was applied to that track from a previous action has fully faded out and that it can now be used by the system to apply the sets of future actions too (i.e., it becomes a free track).

Figure 12.3 shows the basic ‘blending’ transition that will occur when a set is applied to the animation controller which shares the same group name with an animation set already set on the mixer. We are showing the blending case here, but as you will see, blending is an optional property we can specify for any animation set in the .act file.

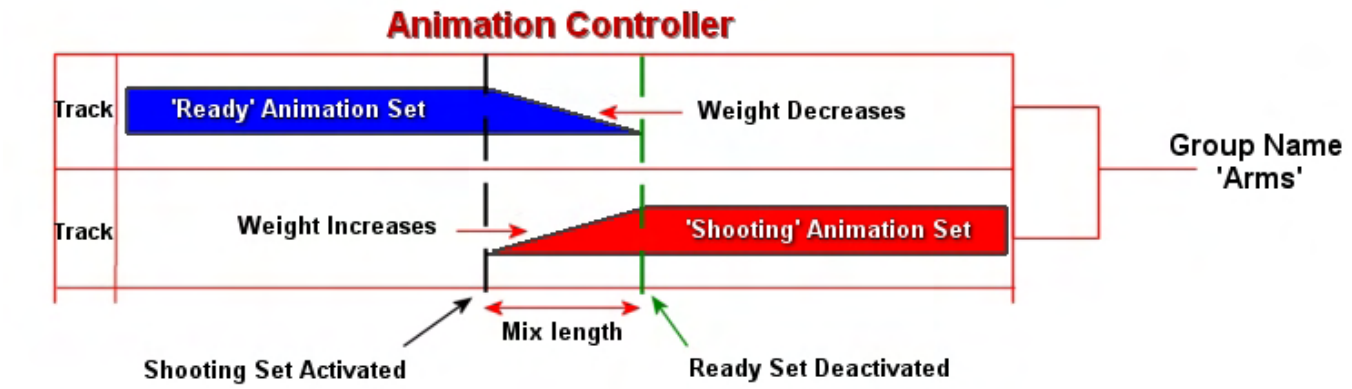


Figure 12.3

In Figure 12.3 it is assumed that the ‘Ready’ animation set and the ‘Shooting’ animation set belong to two different actions but also belong to the same group (‘Arms’). It is also assumed that the action that was applied first to the actor was the one that contained the ‘Ready’ animation.

In this diagram we show the process that occurs between two animation sets from two different actions when the second action is applied and both animation sets share the same group name. We see that halfway through the playing of the ‘Ready’ animation set a new action is applied which contains the ‘Shooting’ animation. This animation is part of the same group. In the new action we are about to apply, the ‘Shooting’ animation set is configured to blend in over a period of time specified by its Mix Length. The ‘Shooting’ animation set is assigned to a mixer track and a sequencer event is set immediately that will fade that track’s weight from 0.0 to its full weight over the time specified in the Mix Length property. If we assume for the sake of this discussion that the Mix Length property assigned to the shooting animation set is 5 seconds, the contribution of the ‘Shooting’ animation set on the actor’s bone

hierarchy will become stronger over a 5 second period. Furthermore, at this exact time, we also set a sequencer event for any previous tracks which belong to the same group to fade out over the same period of time. In this simple example, the ‘Ready’ animation set belongs to the same group, so it has a sequencer event set for it that will fade its weight from its current value to zero over the next 5 seconds. We will also schedule a sequencer event on the ‘Ready’ track that will be triggered in 5 seconds time to disable the track when the fade out is complete. At this point, we will make a note that the track is ready to be used again.

Although only one action can be applied to an actor at any given time and no single action can contain more than one animation set with the same group name, multiple sets belonging to the same group can all be contributing to the hierarchy simultaneously during these transition phases. If you look at the area between the markers (the vertical dashed lines) in Figure 12.3, you will see that while the action that contains the ‘Shooting’ set is considered to be the current action in use by the system, animation sets from the previous action are fading out over time. We can see that during the Mix Length period, two animation sets (one from the current action and one from the previous action) are being used to animate the bone hierarchy simultaneously.

At a point halfway through the Mix Length period, the two sets would have an equal contribution to the hierarchy. We can think of this as positioning the gun halfway between the ready pose and the shooting pose (see Figure 12.4). Therefore, as the weight of the ‘Ready’ pose diminishes and the weight of the ‘Shooting’ pose increases, the arms of our actor will slowly move from the ready pose to the shooting pose and we have a transition animation without the artist ever having to create a custom one. You might also imagine that if you applied several actions to the actor over a very short period of time, which each had fading animation sets belonging to the ‘Arms’ group, after the last action had been applied, you would have one animation set for that group fading in, and multiple animation sets (from the various previous actions) in various stages of fading out. Finally, you can couple that with the fact that multiple actions can contain multiple animation sets, each belonging to multiple shared groups, and we have a lot of flexibility here (and a lot of work to do). Of course, we will have to make sure that we have sufficient spare mixer tracks to handle all the transitioning out we will need to perform when new actions are applied and older actions are still fading out.

This same system can be used to transition between animation sets belonging to the ‘Legs’ group as well. When an action is applied to animate the legs of the character in a running motion, this set could be faded in over a period of time. If the animation set for the ‘Legs’ group from a previous action was a walking sequence, the legs would transition from a walk to a run over a period of time, instead of instantly snapping from one to the other; very nice indeed.

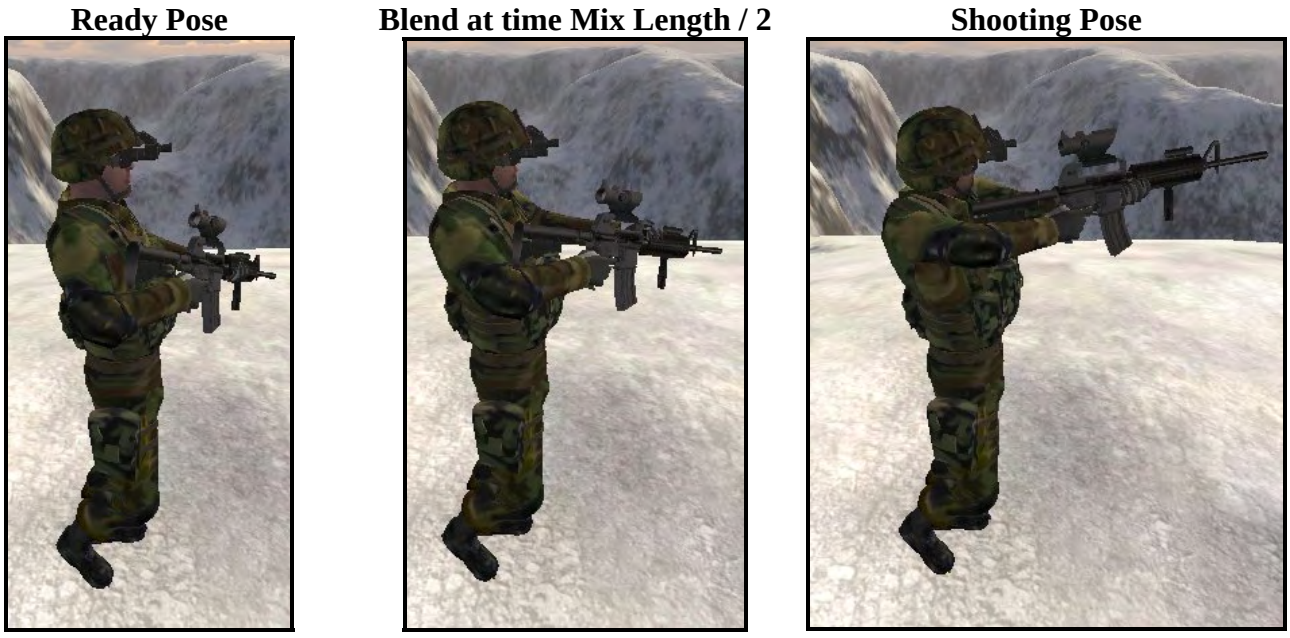


Figure 12.4 : Transition through animation set blending

We will need a way to synchronize the timers for the various tracks when applying actions. For example, if we wish to play the 'Running' animation on the legs and we are currently playing a 'Walking' animation, we would not want the running animation to start from the beginning each time. If the walking animation was halfway through when the running animation set was activated, the two clocks would be wildly out of sync. We would definitely not want the left leg of the character halfway through its walk pose to snap back to its starting position in the run pose. Therefore, in our .act file we will also be able to specify how the clock should be set when an animation set is applied to a mixer track. For example, when applying the running animation to the legs, we would like to set the track position to that of the walk animation set's track that is about to fade out. In other words, we want the new track to take its time from a currently playing track from the same group. That way, the running animation will gracefully take over from the walking animation over the Mix Length period.

As you will see however, all of this functionality is optional. We can have sets defined in actions that are not set to blend or transition. When such animation sets need to be played, any animation sets currently assigned to the controller that share the same group name will be deactivated instantly and the new animation set will take its place. We will also have the option not to synchronize the track clock of a newly assigned animation set to that of the group to which it belongs and may wish to simply play it from the beginning.

Before we get into the discussion of the implementation details and start looking at the contents of an .act file, we have assembled a small glossary of terms that we have collected so far so that there is no confusion as we move forward.

The Action System – Glossary of Terms

ACT File
This is the primary descriptor file which contains the information required to set up the controller tracks for playing out the specified actions. It contains the definitions of one or more actions. Since these are simple text files, we can create them in any text-base editor (e.g., Notepad).

Action
Some combination of animations that convey a higher level animation concept we wish to perform. For example, we may want our actor to swim, jump, run and walk. Each of these actions may need to be comprised of many different animation sets under different circumstances. An action (in the terms outlined by this system) is that collection of animation set names and their properties which describe the components required to carry out the higher level performance.

Every action has a name (e.g., "Walk and Shoot"). It is this name that our application will pass to the CActor::ApplyAction method in order to configure the controller to play that action. The action is comprised of one or more set definitions.

Set Definition
Each action stores one or more set definitions. An individual set definition describes how we would like a single animation set (one set out of a possible many required to perform an action) to be processed and initialized when it is applied to an animation controller mixer track. The various properties stored within a set definition are items such as the name of the animation set to apply, the speed we would like it to play, how much it contributes to the overall animation output (its weight) and other timing and blending properties.

Blending
The action system includes a feature called blending which is primarily used to describe how the individual animation sets should transition between one another when switching between different actions.

Currently, there are only two options for blending: `Off` – which tells the action system that no blending should occur when the animation set is assigned to the controller, and `MixFadeGroup` – which causes a previous set's weight (belonging to the same group) to be decreased, while the new set's weight is increased. This provides a fade-in/fade-out effect between actions.

For many types of actions (although certainly not all) this feature is extremely useful. In many cases, even if we only provide one frame of animation per pose, we can smoothly transition the limbs of a character between those poses without the artist having to provide transition animations. Good candidates are generally animations where the bones are not in wildly different configurations. For example some good blend candidates would typically include Stop to Walk, Stop to Run, Stop to Strafe, Walk to Run, Walk to Strafe, Run to Strafe, Stand to Crouch, (and the reverse for all of these). When the skeletal configurations are very different, blending will not always be a viable solution. For example, going from a swim animation to a walk animation would probably not work all that well and some form of hand crafted transition animation would probably be in order (or just an immediate switch).

While we give up some of the control that hand animated transitions provide, blending usually provides reasonably realistic transitions, with the added advantage of *greatly* reducing animation asset requirements (simple single frame poses will often suffice).

Groups

The AAS includes a concept of groups. A group is essentially just a name that we assign to a set definition describing a relationship between animation sets in different actions. This concept grows out of the requirement for the system to understand which animation set you would like to transition *from*, during the blending process.

Groups are defined between actions, rather than being internal to any one action. That is, the grouping is required for steps that are taken when switching to another action. As a result, set definitions belonging to different actions can be grouped, while within any one action only one set definition may belong to a particular group.

For example, imagine that we have two animation sets named ‘Walk’ and ‘Run’ which contain keyframes for the legs of the character only. In this example, we may define two separate actions for the act of running and walking separately. When we switch between these two actions (and we would like the legs to blend smoothly between the two) the system needs to know that the ‘Walk’ animation set is the one that should be blended out. Because these two set definitions would belong to the same group (perhaps with the name ‘Legs’) it is able to make this determination.

In addition to blending, the group concept is also applied to the various timing properties available via the set definition. These will be explained later.

Now that we know the definitions for the various components of the system, let us have a look at the format of the .act file.

The ACT File Specification

Below is the ACT file specification. Any dynamic properties are surrounded in braces (*i.e. {Label/Restrictions}*), while optional values (depending on certain other values used) are shown in ***bold***.

```
 [General]
ActionCount = {0…n}

; ---------------------------------------------------------------------------
; Action Block
; Repeat for each action, the number of which is defined in the
; above 'ActionCount' variable (relative to 1).
; ---------------------------------------------------------------------------
[Action{1…n}]

Name            = {Action Name}
DefinitionCount  = {0…n}


; ---------------------------------------------------------------------------
; Definition Block
; Repeat for each definition, the number of which is defined in the
; above 'DefinitionCount' variable.
; ---------------------------------------------------------------------------
Definition[{0…(DefinitionCount-1)}].SetName      = {Animation Set Name}
Definition[{0…(DefinitionCount-1)}].GroupName  = {Group Name (Unique to this action)}
Definition[{0…(DefinitionCount-1)}].Weight         = {Track Weight}
Definition[{0…(DefinitionCount-1)}].Speed          = {Track Speed}
Definition[{0…(DefinitionCount-1)}].BlendMode   = {Off | MixFadeGroup}
Definition[{0…(DefinitionCount-1)}].TimeMode    = {Begin | MatchGroup | MatchSpecifiedGroup}

Definition[{0…(DefinitionCount-1)}].MixLength          = {Seconds to Blend if MixFadeGroup}
Definition[{0…(DefinitionCount-1)}].TimeGroup        = {Group Name if MatchSpecifiedGroup}
; ---------------------------------------------------------------------------
; End Definition Block
; ---------------------------------------------------------------------------




; ---------------------------------------------------------------------------
; End Action Block
; ---------------------------------------------------------------------------
```

At the top of file is the General block, which describes properties global to the file. In this block there is just a single property we need to specify -- the number of action definitions that are going to follow. Each action definition contains the data for a single action. If the ActionCount property in the General block is assigned a value of 10, this should be followed by 10 Action blocks.

Following the General block are the Action blocks. Each action block should have the name Action*N* where *N* is the number of the block currently being defined. The number of action blocks should be equal to the ActionCount property specified in the General block.

Each action block has two properties that need to be set -- the name of the action and the number of set definitions it will contain. The name assigned to the action is the same name the application will use to apply that action to the actor, so you should use something descriptive (e.g., "Strafe and Shoot"). The DefinitionCount property is where we specify the number of animation sets that will comprise this action. If we set this value to *N*, it should be followed by *N* Definition blocks nested inside the Action block. Each definition contains the name of an animation set and the properties needed to assign it to the animation mixer.

Each Definition block inside an Action block has a name in the format Definition*N*, where *N* is the numerical index of the definition inside the Action block. Unlike the General and Action blocks, each definition block has many properties to set which describe how the animation set is set up.

We will now discuss the properties of the definition block.

```
Definition[{0…(DefinitionCount-1)}].SetName      = {Animation Set Name}
Definition[{0…(DefinitionCount-1)}].GroupName    = {Group Name (Unique to this action)}
Definition[{0…(DefinitionCount-1)}].Weight       = {Track Weight}
Definition[{0…(DefinitionCount-1)}].Speed        = {Track Speed}
Definition[{0…(DefinitionCount-1)}].BlendMode    = {Off | MixFadeGroup}
Definition[{0…(DefinitionCount-1)}].TimeMode     = {Begin | MatchGroup | MatchSpecifiedGroup}
Definition[{0…(DefinitionCount-1)}].MixLength    = {Seconds to Blend if MixFadeGroup}
Definition[{0…(DefinitionCount-1)}].TimeGroup    = {Group Name if MatchSpecifiedGroup}
```

Set Name

This is the name of the animation set for which this definition is applicable. It should match the name of an animation set in the X file. If you have created the art assets yourself, then you will know the names you have given to your sets, otherwise your artist will supply these names to you. If you are using models that you have downloaded off the internet then you can always convert the X file to a text based X file, open it up in a text editor and search for the Animation Set data objects to retrieve their names.

Group Name

This is the where you specify the group name you would like to apply to this set definition. There should not be another definition in this action which belongs to the same group. Animation sets in different actions that belong to the same group can be transitioned between when a new action is applied. For example, you might have one definition in every action that animates the legs of the character. You might give each of these definitions in each action the group name of 'Legs'.

Weight

This is where we specify the weight of the animation set and thus control the strength of its contribution to the actor's hierarchy. You will usually set this to 1.0 so that the animation, when fully transitioned in, will play at the strength originally intended by the asset creator.

Speed

This is where we specify the speed at which we would like the animation set play when it is first activated. The speed of the track can still be modified by the application after the action has been applied, and we do that very thing in Lab Project 12.3 to speed up the walking animation based on the character's current velocity.

BlendMode

This property can be set to one of two values; Off or MixFadeGroup.

Off

If set to Off, this animation set will not transition to full strength over a period of time and will not be blended with any other animation sets currently playing that have been assigned to the same group. When the action is applied, any sets with the Off BlendMode state will be immediate assigned to a free track at the weight and speed described above. If the previous action was playing an animation set that belonged to the same group, that animation set will be deactivated immediately. There will be no transition from one to the next.

MixFadeGroup

When the BlendMode property is set to MixFadeGroup, we are stating that when this action is applied, this animation set's weight should slowly transition to its full strength (specified by the weight property above) over a period of time. At the same time, if the previous action was playing an animation set that has been assigned the same group name, then that animation set will have its weight configured to fade out over this same period of time. During this transition time, both the animation sets will be active and blended together to create transition poses from one animation set to the next. Once the previous animation set has had its weight faded out to zero, it is deactivated and the track it was assigned to is free to be used by the system again.

Mix Length

This property is only used if the animation set's BlendMode property has been set to MixFadeGroup. It specifies how long we would like this definition to take to transition to full strength. Similarly, it tells us how long any current animation set belonging to the same group should take to transition out.

TimeMode

The TimeMode property allows us to configure the starting track time for an animation set when it is first applied to the mixer. This is a very useful property that allows us to synchronize the track time of the newly applied animation set with the track time of either a previous animation set from the same group that is transitioning out, or to that of a track playing an animation set belonging to a different group. The latter is also very important as it allows us to synchronize the arms and legs animations even though they belong to different groups. We might imagine that the arms animation set is configured to take its time from the legs animation set, so even when we apply a new animation set to the torso (such as walking), it will take its starting time from the current position of the legs so that the walking movement of both the arms and legs are synchronized.

There are three possible values for this property that our action system understands. They are listed below.

Begin

No time matching is performed. When this animation set is applied, the track clock is set such that the animation set starts playing from the beginning. In this mode, the animation set is not synchronized with any other animation sets in the same action and operates in isolation from a timing perspective.

MatchGroup
This mode is extremely useful when an animation set is configured to blend. It informs the action system that when this animation set is applied, we do not want to start playing it from the beginning but instead, its periodic position should match the current periodic position of an animation set currently playing (from a previously set action) that belongs to the same group. If we imagine two actions, where one has a 'Walk' animation and another has a 'Run' animation, if both of these were assigned to the legs group, the 'Run' animation could inherit the time currently being used by the 'Walk' set to assure a smooth transition.

MatchSpecifiedGroup
This mode is used to ensure synchronization between animation sets within an action that belong to different groups. As an example, we may apply an action to a character that contains an 'Arms' group and a 'Legs' group. The 'Legs' group might be set to MatchGroup mode so that it inherits the time from a current 'Legs' group set to ensure a smooth transition. We might then configure the 'Arms' group in our new action to inherit the time from the 'Legs' group so that the animation sets for both the arms and legs not only remain synchronized with each other, but also inherit their periodic position from a 'Legs' group that was playing in the previous action.

If this mode has been set for a given animation set, the name of the group that you would like this animation set to take its timing information from must also be specified using the TimeGroup property.

Time Group
This property is needed only for definitions that have their TimeMode property set to MatchSpecifiedGroup. It should contain the name of the group you would like this animation set to inherit its initial track time from.

We have now covered the complete specification of our new .ACT file format and have examined what each property does. This will go a long way towards helping us understand the implementation of our new animation system. While the .ACT file specification does not provide as much flexibility as a scripting system -- where we would be able to include conditional branches based on certain states of the application for instance -- this simple specification does provide us with enough control to be able to achieve a reasonably realistic set of actions and transitioning effects for use in our current demos.

An .ACT file example

Looking at an example should help to clarify the subject. Below you can see a very simple .act file that contains three actions. Each action contains three set definitions. They describe animation sets that work with the torso, legs and hips. The hips animations were created by the artist to counteract the rotation of the pelvis caused by the leg animations. That is, the hips animation essentially undoes the rotation to the midriff caused by the leg animations. This stops the torso from rotating side to side as the legs walk.

Note: Lines that start in an .ini file with a semicolon are ignored by the loading functions. They are comments and equivalent to the C++ double forward slash.

The three actions defined in this .ACT file are ‘Walk_At_Ease’, ‘Walk_Ready’ and ‘Walk_Shooting’. Each action contains three set definitions for the legs, hips and torso. The legs, hips and torso definitions for each action are assigned to the ‘Legs’, ‘Hips’, and ‘Torso’ groups, respectively.

```
; Action Definition Script 'US Ranger.act'
[General]
ActionCount = 3

; ------------------------------------------------------------------------------
; Name : Walk_At_Ease
; Desc : Standard walking with weapon down by side.
; ------------------------------------------------------------------------------
[Action1]

Name                        = Walk_At_Ease
DefinitionCount             = 3

Definition[0].SetName       = Walk_Legs_Only
Definition[0].GroupName     = Legs
Definition[0].Weight        = 1.0
Definition[0].Speed         = 1.0
Definition[0].BlendMode     = MixFadeGroup
Definition[0].MixLength     = 0.3
Definition[0].TimeMode      = MatchGroup

Definition[1].SetName       = Walk_Variation
Definition[1].GroupName     = Hips
Definition[1].Weight        = 1.0
Definition[1].Speed         = 1.0
Definition[1].BlendMode     = Off
Definition[1].TimeMode      = MatchSpecifiedGroup
Definition[1].TimeGroup     = Legs

Definition[2].SetName       = Gun_At_Ease
Definition[2].GroupName     = Torso
Definition[2].Weight        = 1.0
Definition[2].Speed         = 1.0
Definition[2].BlendMode     = MixFadeGroup
Definition[2].MixLength     = 1.0
Definition[2].TimeMode      = MatchSpecifiedGroup
Definition[2].TimeGroup     = Legs

; ------------------------------------------------------------------------------
; Name : Walk_Ready
; Desc : Standard walking with weapon at the ready (clasped in both hands)
; ------------------------------------------------------------------------------
[Action2]

Name                        = Walk_Ready
DefinitionCount             = 3

Definition[0].SetName       = Walk_Legs_Only
Definition[0].GroupName     = Legs
Definition[0].Weight        = 1.0
Definition[0].Speed         = 1.0
```

```
Definition[0].BlendMode     = MixFadeGroup
Definition[0].MixLength     = 0.3
Definition[0].TimeMode      = MatchGroup

Definition[1].SetName       = Walk_Variation
Definition[1].GroupName     = Hips
Definition[1].Weight        = 1.0
Definition[1].Speed         = 1.0
Definition[1].BlendMode     = Off
Definition[1].TimeMode      = MatchSpecifiedGroup
Definition[1].TimeGroup     = Legs

Definition[2].SetName       = Gun_Ready_Pose
Definition[2].GroupName     = Torso
Definition[2].Weight        = 1.0
Definition[2].Speed         = 1.0
Definition[2].BlendMode     = MixFadeGroup
Definition[2].MixLength     = 1.0
Definition[2].TimeMode      = Begin



; ------------------------------------------------------------------------------
; Name : Walk_Shooting
; Desc : Standard walking with weapon raised in a shooting pose.
; ------------------------------------------------------------------------------
[Action3]

Name                        = Walk_Shooting
DefinitionCount             = 3

Definition[0].SetName       = Walk_Legs_Only
Definition[0].GroupName     = Legs
Definition[0].Weight        = 1.0
Definition[0].Speed         = 1.0
Definition[0].BlendMode     = MixFadeGroup
Definition[0].MixLength     = 0.3
Definition[0].TimeMode      = MatchGroup

Definition[1].SetName       = Walk_Variation
Definition[1].GroupName     = Hips
Definition[1].Weight        = 1.0
Definition[1].Speed         = 1.0
Definition[1].BlendMode     = Off
Definition[1].TimeMode      = MatchSpecifiedGroup
Definition[1].TimeGroup     = Legs

Definition[2].SetName       = Gun_Shooting_Pose
Definition[2].GroupName     = Torso
Definition[2].Weight        = 1.0
Definition[2].Speed         = 1.0
Definition[2].BlendMode     = MixFadeGroup
Definition[2].MixLength     = 1.0
Definition[2].TimeMode      = Begin
```

In each action, the same animation set for the legs group is being used. This is the animation set with the name ‘Walk_Legs_Only’. As an action is applied, the track position for the ‘Walk_Legs_Only’ animation set in that action will take its timing from a previous set assigned to the same group that is currently playing. In this simple example, where each action uses the same animation set for the legs, it simply means that as the ‘Walk_Legs_Only’ animation set is set on the mixer for a given action, its initial position will be set to the ‘Walk_Legs_Only’ set that was being played by the previous action. The ‘Legs’ group in each action is also set to MixFadeGroup so that the animation set will fade in over the period specified in the MixLength parameters (0.3 seconds in this example). In this particular .ACT file, all actions use the same animation set for the legs, so the transition is a mostly futile. However, we might want to add an additional action such as ‘Idle_At_Ease’ which would use a different animation set for the legs; perhaps one in which the character was no longer walking. In such a case, the MixFadeGroup property would ensure that when we transition from the ‘Idle_At_Ease’ action to the ‘Walk_At_Ease’ action, there would be a 0.3 second transition while the legs move from the idle pose into the walking pose. However, if we know in advance that we only wanted to use walking poses (unlikely) then we could set the BlendMode to ‘Off’.

The ‘Hips’ group animation set in each action does not use blending. Once again, it is the same set being used by the ‘Hips’ group in each action. As we discussed in the previous paragraph, in this section we are assuming that there is only one ‘Hips’ animation set used by all actions, so there is no need to blend between ‘Hips’ animation sets in different actions. In each action however, the ‘Hips’ group animation set is configured to take its timing from the ‘Legs’ group. This is very important in this instance because the ‘Hips’ animation set is really nothing more than an inverse transformation applied to the pelvis to undo any rotation applied to it by the walking of the legs. That is, the artist had to rotate the pelvis bone to make the legs animate properly, but this inverse transformation is applied to the next bone up to stop the pelvis rotation (which is the parent bone) being reflected up to the torso.

Finally, the last animation set specified in each action is the one that belongs to the ‘Torso’ group. A different animation set is used in each action for this purpose. In the first action (‘Walk_At_Ease’), the ‘Torso’ animation set is called ‘Gun_At_Ease’ and it animates the torso such that the character is walking with his gun in his hand, with his arms swinging back and forth in time with his legs. Because of this, in the first action, the timing of this animation set is taken from the ‘Legs’ group so that the swinging arms are synchronized with the walking legs.

In the second action, the animation set in the ‘Torso’ group is called ‘Gun_Ready_Pose’. This adjusts the torso of the character so that he is holding the gun to his chest and looking alert, but not yet assuming a hostile posture. Because this animation no longer needs to be synchronized with the movements of the legs (the arms are no longer swinging back and forth), the timing mode of this animation is set to ‘Begin’. It will begin playing from the beginning as soon as this action is activated. Notice however that it is still set to blend with other sets in its group. This means, if the previous action playing was ‘Walk_At_Ease’, when this new action is applied, the arms would slowly transition from swinging back and forth up to the ready position over a period of one second.

The third action has a ‘Torso’ group animation set called ‘Gun_Shooting_Pose’. This is a single frame animation of the torso holding the gun up to his face in an aiming posture. Since this pose requires no relationship with the animations being played out on other parts of the character’s body, its time mode is set to ‘Begin’. It is also set to blend in from animation sets within the same group. That is, if this action

is applied and the previous action being played was ‘Walk_Ready’, the torso of the character would transition from the ready pose to the shooting pose over a period of one second.

We have now covered all the properties that our Action Animation System will have to work with. We have also seen the format in which the data will be presented to our application. We now know how to write .ACT files in a text editor and the format in which the contents need to be laid out. So let us now begin coding the Action Animation System.

The Action Animation System – Implementation

The structures we will add to CActor will very much mirror the layout of the *.ini* file (.ACT file). The actor will maintain an array of CActionDefinition structures and each action definition structure will contain the information for a single action loaded from the .ACT file. A LoadActionDefinitions function will be added to the CActor interface. This method, which will be called from the application and passed the name of an .ACT file, will load the data and store it internally. For each action definition in the .ACT file, a CActionDefinition structure will be allocated, populated, and added to the actor’s CActionDefinition array.

Each CActionDefinition structure will maintain an array of SetDefinition structures. Each set definition will contain the data for a single animation set when the action is applied. As you can see, this follows the exact layout of the .ACT file. The .ACT contains a number of action definitions and each action definition contains a number of set definitions. In code, the actor contains an array of action definitions and each action definition contains an array of set definitions. The ApplyAction method will also be added to the interface of CActor so that the application can inform the actor to apply one of its actions at any time.

Of course, simply maintaining an array of the information loaded from the .ACT file is not enough by itself to describe the state that the controller may be in at any given time to our action system. For example, we will need to know, for each track on the mixer, is there currently an animation set assigned to it or is it free for our system to use? Is the track in the phase of transitioning out? What was the name of the animation set assigned to that track and is it used by the current action we are applying? A support structure of some kind needs to be introduced. We will need this anyway for us to quickly ascertain during the blending process, which group the animation set assigned to a given track belongs to. We know for example that if the time mode of the animation set we are about to assign has been set to MatchGroup, we will need to search the tracks of the animation mixer for an animation set that belongs to the same group. As the mixer has no concept of groups, we will obviously need a support structure that will store this information for each track.

The support class we will use is called CActionStatus. This will be a new member of CActor which contains the status of each track on the animation mixer. Internally, the CActionStatus class will manage an array of TrackActionStatus structures; one for each track on the mixer. Each TrackActionStatus structure will contain the current properties for a given track on the animation mixer.

To clarify, if the actor had a 15 track animation mixer, the actor would contain a single CActionStatus member that would contain an array of 15 TrackActionStatus structures. Each TrackActionStatus structure contains information about whether its associated track is currently in use by the action system, the name of the set currently assigned to it, the group that set belongs to, and so on.

The following diagram shows the relationships between CActor and the various structures just discussed.

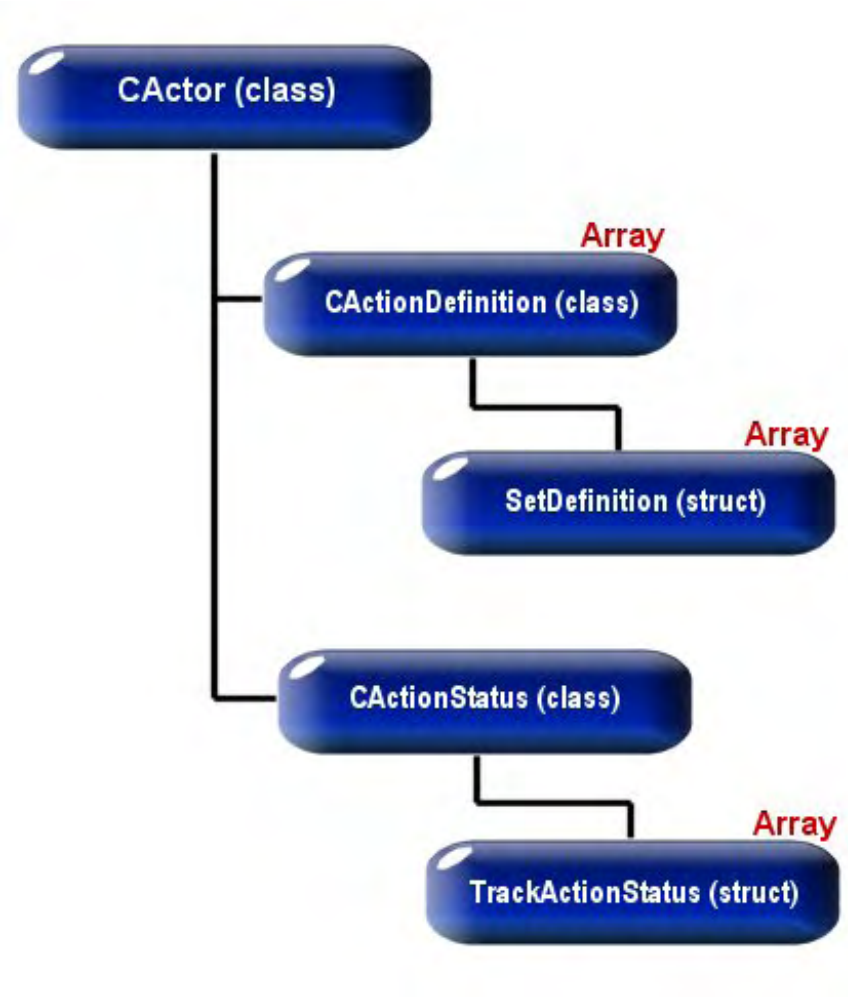


Figure 12.5

Notice that there is a single CActionStatus member that contains the status of the current controller being used by the actor. As the controller is essentially swapped out when referencing is being used, we will also have to do the same with the CActionStatus object. That is, each CObject reference to an actor will now contain not only its own animation controller but also its own CActionStatus object describing the status of all the tracks on its controller's mixer. Our actor's AttachController method will now accept both a pointer to the controller we wish to attach and the CActionStatus object that describes the status of that controller's tracks. This will be a very small adjustment which we will discuss later.

The CActionDefinition array will be allocated when the user calls the CActor::LoadActionDefinitions function. There will be an element in this array for each action defined in the file. Let us now have a look at the changes to CActor.

The CActor Class – Code Changes

We have added some new member variables to CActor to store the data of the action system. As our CActor class definition is now getting quite large, we have only shown the new or modified variables and functions that we are going to introduce in this workbook.

Excerpt from CActor.h

```
class CActor
{
…
…
 private:

    // New 'Action' set data variables
    CActionDefinition   *m_pActions;        // An array of loaded actions
    ULONG                m_nActionCount;    // The number of actions
    CActionStatus       *m_pActionStatus;   // Stores the status of all tracks
…                                           // for the action system
…
  public :
    // New 'Action' Methods
        HRESULT           LoadActionDefinitions      ( LPCTSTR FileName );
        HRESULT           ApplyAction                ( LPCTSTR ActionName );
        CActionDefinition * GetCurrentAction         ( ) const;
        CActionStatus     * GetActionStatus          ( ) const;
…
…
    // Existing functions modified by action system
    LPD3DXANIMATIONCONTROLLER DetachController( CActionStatus ** ppActionStatus );

    void AttachController    ( LPD3DXANIMATIONCONTROLLER pController,
                               bool bSyncOutputs = true,
                               CActionStatus * pActionStatus = NULL );

    HRESULT SetActorLimits      (       ULONG MaxAnimSets = 0,
                                        ULONG MaxTracks = 0,
                                        ULONG MaxAnimOutputs = 0,
                                        ULONG MaxEvents = 0,
                                        ULONG MaxCallbackKeys = 0 );
  }
```

Let us first discuss the three new member variables that have been added.

CActionDefinition *m_pActions

This is a pointer to an array of CActionDefinition objects that will be allocated when the application issues a call to the CActor::LoadActionDefinitions method. Each element in this array is a class that manages the data for a single action loaded from the .ACT file. The CActionDefinition class exposes methods of its own, such as LoadAction and ApplyAction, which the actor uses to perform the heavy lifting associated with loading or applying an action. The actor methods are really just lightweight wrappers that make calls to the methods of the current action being loaded or applied. We will look at the members and methods of the CActionDefinition class shortly.

ULONG m_nActionCount
This member is set in the CActor::LoadActionDefinitions method and contains the number of actions defined in the .ACT file. Therefore, it also describes the size of the above array.

CActionStatus *m_pActionStatus
This is a pointer to a single CActionStatus object that will contain the current states of each track on the animation mixer. It is used by the Action System to remember which animation sets have been assigned to which tracks, which groups those animation sets belong to, and the status of those tracks (e.g., if they are in the middle of a transition).

We will now discuss the new methods that have been added to CActor. Since most of these functions are light wrappers around calls to the methods of the CActionStatus and the CActionDefinition objects, we will not get the full picture until these objects have been covered as well. However, it should be fairly obvious from a high level, with a little explanation, what these methods are doing.

CActor::LoadActionDefinitions

This method is called by the application after the actor has been allocated and populated by the mesh and animation data from the X file. This function is called and passed the name of an .ACT file that contains the data for the action system. This function will open that file, allocate a CActionDefinition array large enough to contain the correct number of actions, and then populate this array with the action data from the file. It will also allocate a new CActionStatus object that has a large enough internal TrackActionStatus array to store the status of every track on the animation mixer.

The function is passed the name of the file we wish to load the action definitions from. The first thing the function does is delete any previous CActionDefinitions array that may exist in the actor prior to this call. It also releases its CActionStatus pointer if one already exists. Notice that because our CActionStatus object will have to be attached and detached from the controller and stored in an external object when referencing is being used, we have implemented the COM-style AddRef and Release mechanism for this object.

```
HRESULT CActor::LoadActionDefinitions( LPCTSTR FileName )
{
    ULONG   ActionCount = 0, i;
    HRESULT hRet;

    // Clear any previous action definitions
    if ( m_pActions ) delete []m_pActions;
    m_pActions     = NULL;
    m_nActionCount = 0;

    // Clear any previous action status
    if ( m_pActionStatus ) m_pActionStatus->Release();
    m_pActionStatus = NULL;
```

We will now retrieve the ActionCount property from the .ACT file. Recall from our studies in Module I that we use the Win32 function GetPrivateProfileInt to read in integer data from an .ini text file.

```
    // Retrieve the number of actions stored in the file
    ActionCount = ::GetPrivateProfileInt( _T("General"),
                                          _T("ActionCount"),
                                          0,
                                          FileName );

    if ( ActionCount == 0 ) return D3DERR_NOTFOUND;
```

Just to recap, the first parameter is the name of the data block in the file that we wish to retrieve the value from. Recall from our .ACT format that the action count property is set in the General block. As the second parameter we pass in the name of the property we wish to fetch the value for (ActionCount). If this property is not found, then the third parameter specifies a default value to be used. In this instance, if the ActionCount property is missing we have an invalid .ACT file and return a default action count of 0 which will cause the function to return without loading any action data. As the final parameter, we pass in the name (and path) of the file we wish to extract the value from.

Now that we know how many actions there are, we can allocate the actor's CActionDefinition array to the correct size so that there is an element for each action in the file. We also allocate the CActionStatus object.

```
    // Allocate the new actions array
    m_pActions = new CActionDefinition[ ActionCount ];
    if ( !m_pActions ) return E_OUTOFMEMORY;

    // Allocate the new status structure
    m_pActionStatus = new CActionStatus;
    if ( !m_pActionStatus ) return E_OUTOFMEMORY;
```

The CActionStatus object will maintain an array of TrackActionStatus objects for each track on the mixer. Each one will contain the status of a track. Let us now fetch the number of tracks available on this actor's animation mixer and send this value to the CActionStatus::SetMaxTrackCount method. We will cover this method in a moment, but it essentially allocates the array of TrackActionStatus structures to be as large as described by the input parameter. We will also copy over the action count property that we extracted from the .ACT file.

```
    // Setup max tracks for action status
    m_pActionStatus->SetMaxTrackCount( (USHORT)GetMaxNumTracks() );

    // Store the count
    m_nActionCount = ActionCount;
```

It is now time to load each action in the .ACT file into each element in the CActionDefinitions array we just allocated. To do this, we loop through each action definition in the array and call its LoadAction method. As you can see, it is the CActionDefinition::LoadAction function which actually loads the data from each action in the .ACT file. We pass it two parameters. The first is a numerical index that, when appended to the word 'Action', will describe the name of the Action block in .ACT file that the current action needs to fetch its data from. As the second parameter, we simply pass the file name of the .ACT file that contains the action data.

```
    // Process the actions
    for ( i = 0; i < ActionCount; ++i )
    {
        // Load the definition from file
        hRet = m_pActions[i].LoadAction( i + 1, FileName );
        if ( FAILED(hRet) ) return hRet;

    } // Next Action in file

    // Success!
    return S_OK;
}
```

And that is it for this function. Obviously, many of the blanks will be filled in when we cover the code to the CActionDefinition methods and the CActionStatus methods. However, by examining the actor's interaction with these objects first, we will have a better understanding of their place within the entire system when we do get around to covering them.

CActor::GetCurrentAction

The next CActor method we will cover is a useful function should the application wish to query which action is currently being played by the actor. Examining the code also establishes the responsibility of the actor's CActionStatus member to keep a record of the current action that has been applied.

```
CActionDefinition * CActor::GetCurrentAction( ) const
{
    // Return the current action if status is available
    if ( !m_pActionStatus ) return NULL;
    return m_pActionStatus->GetCurrentAction();
}
```

Notice that it is the CActionStatus structure that stores the current action (the action that was last applied) and this property can be fetched via CActionStatus::GetCurrentAction. The CActionStatus object will be covered in a moment, but is really little more than a storage container with functions to set and retrieve its data.

CActor::GetActionStatus

This function allows the application to fetch a pointer to the CActionStatus object being used by the actor. This is useful since the CActionStatus object contains the status information for each track of the animation controller. If the application wanted to play an additional animation that was not part of the action, it could fetch the CActionStatus object, use it to find a free track, and flag that track as being in use. If we did not do this, then the action system would have no way of knowing that one of the tracks on the animation mixer was currently being used by an animation set external to the action system and erroneously think it could use that track the next time an action was applied.

```
CActionStatus * CActor::GetActionStatus( ) const
{
    // Validate requirements
    if ( !m_pActionStatus ) return NULL;

    // Add a reference to the action status, we have a floating pointer
    m_pActionStatus->AddRef();

    // Return the pointer
    return m_pActionStatus;
}
```

Another reason to fetch the CActionStatus object is to store it in a CObject. That is, when multiple CObjects reference the same CActor, each object will contain its own controller and status object that will be set before that object is animated.

CActor::ApplyAction

This next function is called by the application when it wishes to change the current action being played to a new one. All the application has to do is call this function and specify the name of the action it wishes to apply. This will be the name assigned to one of the actions in the .ACT file.

This function is the primary runtime interface to the Action Animation System. Once the action data has been loaded, it is the only function that needs to be called to apply new actions to the controller. This function is rather small, since most of the core work takes place inside the CActionDefinition::ApplyAction method (just as most of the loading of data was performed inside this object) which we will examine later.

The first thing this function does is test to see that the actor has an animation controller and a CActionStatus object defined. If not, it returns, since there is no controller to configure with action data. We then get a pointer to the current action being played and test to see if its name is the same as the name passed in. If it is, then the application has asked for an action to be played which is already playing, and there is nothing to do. In this case we will simply return.

```
HRESULT CActor::ApplyAction( LPCTSTR ActionName )
{
    ULONG   i;
    HRESULT hRet;

    // Bail if there is no active controller
    if ( !m_pAnimController || !m_pActionStatus ) return D3DERR_INVALIDCALL;

    // If this action is already set, ignore
    CActionDefinition * pCurrentAction = m_pActionStatus->GetCurrentAction();

    if ( pCurrentAction && _tcsicmp( pCurrentAction->GetName(), ActionName )== 0 )
        return D3D_OK;
```

Notice that it is the CActionStatus object which contains a pointer to the current CActionDefinition that has been applied (we can get a pointer to it via its GetCurrentAction method). Once we have a pointer to the current CActionDefinition, we can use its GetName function to retrieve a string containing its name. In the above code, we use the _tcsicmp string comparison function to compare the name of the current action with the name of the action the application would like to apply. If the _tcsicmp function returns zero, the two strings are identical and there is no need to do anything.

In the next section of code we have determined that the application would like to set a new action. The first thing we must do is loop through each action in the actor's CActionDefinition array and search for the one with a name that matches the action name passed into the function. Here is the remainder of the function:

```
    // Loop through each action
    for ( i = 0; i < m_nActionCount; ++i )
    {
        CActionDefinition * pAction = &m_pActions[i];

        // Compare the name, and apply if it matches
        if ( _tcsicmp( ActionName, pAction->GetName() ) == 0 )
        {
            // Apply the action
            hRet = pAction->ApplyAction( this );
            if ( FAILED(hRet) ) return hRet;

            // Store the action
            m_pActionStatus->SetCurrentAction( pAction );

            // Success!
            return D3D_OK;

        }  // End if names match

    } // Next action

    // Unable to find this action
    return D3DERR_NOTFOUND;
}
```

Once we find a matching action in the CActionDefinition array we will use its ApplyAction method to apply the action to the controller (covered in a moment). This is obviously where the core work of applying the animation sets to the controller and setting blending sequences will take place. After we have applied the current action, we must let the CActionStatus object know that there is a new current action. We do this using the CActionStatus::SetCurrentAction method. This method just stores the passed pointer in a member variable. As discussed, the CActionStatus object is just storage object that has its properties set and retrieved via a few member functions.

Before we get to the core code inside the CActionDefinition object, we will first discuss the layout and methods of the CActionStatus object since it is used extensively by the CActionDefinition::ApplyAction function.

The CActionStatus Class

The CActionStatus class stores the properties of each track on the animation mixer. For a given controller only one CActionStatus object will exist. Internally, the object has only four member variables. It stores the number of tracks on the controller, the current action that is being used by the controller, its own reference count (it uses COM-style lifetime encapsulation semantics), and a pointer to an array of TrackActionStatus structures. There will be one of these structures for each available track on the animation mixer and each structure stores five bits of information about a given track.

Below is the class definition for CActionStatus. Notice that the TrackActionStatus structure is defined in the CActionStatus namespace. Take a look at the member variables and functions that are specified as part of this namespace and also notice that some of the simpler methods are inline.

Excerpt from CActor.h

```
class CActionStatus
{
public:

    struct TrackActionStatus // Status of a single track
    {
        bool   bInUse;             // This track is in use
        bool   bTransitioningOut;  // The track is currently transitioning out
        TCHAR  strGroupName[127];  // The name of the group assigned to this track
        TCHAR  strSetName[127];    // The name of the set assigned to this track

        const CActionDefinition * pActionDefinition;  // The action definition
                                                      // to which this track
                                                      // applies

         ULONG                    nSetDefinition;     // The set definition index,
                                                      // of the above action, to
                                                      // which this track applies
    };

    // Constructors & Destructors for This Class.

    CActionStatus( );
    virtual ~CActionStatus( );

    // Public Functions for This Class.

    ULONG               AddRef           ( );
    ULONG               Release          ( );
    HRESULT             SetMaxTrackCount ( USHORT Count );
    void                SetCurrentAction ( CActionDefinition * pActionDefinition);
    USHORT              GetMaxTrackCount ( ) const  { return m_nMaxTrackCount; }
    TrackActionStatus * GetTrackStatus   ( ULONG Index ) const
                                          {
                                            return ( Index < m_nMaxTrackCount ) ?
                                            &m_pTrackActionStatus[ Index ] :
                                            NULL;
                                           }
```

```
    CActionDefinition * GetCurrentAction ( ) const   { return m_pCurrentAction; }

private:

    // Private Variables for This Class.

    USHORT              m_nMaxTrackCount;     // The maximum tracks for controller
    TrackActionStatus * m_pTrackActionStatus; // Array of track status structures
    CActionDefinition * m_pCurrentAction;     // The current action applied
    ULONG               m_nRefCount;          // Reference count variable
};
```

This class has only four member variables, so let us discuss them before we look at the functions.

USHORT m_nMaxTrackCount
This member will contain the number of tracks on the animation mixer. This value is set when the actor calls the CActionStatus::SetMaxTrackCount method when the actor is first loaded, or when the application resizes the limits of the actor's controller using CActor::SetActorLimits.

Any time the actor clones its controller and extends the number of tracks, the CActionStatus::SetMaxTrackCount method will be called so that the m_nMaxTrackCount member can have its value updated and the TrackActionStatus array (discussed below) can be resized to contain this many elements.

TrackActionStatus * m_pTrackActionStatus
This member is a pointer to an array of TrackActionStatus structures. There will be one TrackActionStatus structure in this array for every track on the animation mixer. That is, the size of this array will be equal to the value of m_nMaxTrackCount discussed above.

Just like the previous value, this array has its size set in the CActionStatus::SetMaxTrackCount method, which is called by the actor whenever the controller is first created or its maximum track limits are extended (via cloning).

Each TrackActionStatus structure in the array contains five very important pieces of information about each track that our Action System will need to know in order to apply actions successfully. Let us discuss the members of this structure.

bool bInUse
This is a simple boolean member which specifies whether the associated mixer track is currently in use. When an action is applied, the animation sets referenced by that action will need to be assigned to tracks on the animation mixer. The action system will need to know which tracks are currently free and can have those sets assigned to them.

If this member is set to false then the track is free to have an animation set assigned to it. If it is set to true then it means it is being used or was being used by a previous action and is still in the process of having its weight faded out.

bool bTransitioningOut

This boolean is set to true if the associated track is currently in the process of being faded out. When an action is applied, and one of the sets in that action shares a group name with a previously assigned animation set, the previous animation set will be set to transition out over a period of time if blending is enabled for the new set. The previous set of the same group name will have a sequencer event activated for it that will fade the track's weight to zero over a specified period of time. This boolean will then be set to true so that the action system knows that this animation set and its track must be left alone to fade out. A sequencer event will also be set for the track at the end of the fade out period that disables the track on the mixer. The next time a new action is applied, we can search the tracks that have this boolean set to true and if we discover that its associated track has been disabled, we can once again mark this track as no longer in use. This is done by setting the bInUse Boolean and the bTransitioningOut Boolean to false.

The need for this boolean will become clear when we cover the code to the CActionDefinition::ApplyAction function. Essentially, it is our way of saying that this animation set was at some time in the past set to transition out over a period of time so leave it alone until that is done, regardless of what other actions may be applied in the fade out period and what their blend settings may be. Once an animation set has been configured to fade, it essentially gets left to its own devices by the action system until the fade out is complete. Then the track can once again be used by future actions.

TCHAR strGroupName[127]

As discussed earlier, one of the key properties for the blending process between actions is the group name assigned to an animation set used by those actions. If an action is applied which contains an animation set that shares the group name with an animation set in the previously active action, and the animation set in the new action has been configured to blend, the action system will fade the previous animation set out while simultaneously fading the weight of the new animation set in.

The action system performs this task by recognizing that for any action there should only be a single track from the same group. Before an animation set is assigned, we first search the tracks of the mixer to see if a track existed in the previous action that shares the same group. If the new action is configured to group blend, the previous track will be faded and the new track introduced over the same period of time. Obviously, for the action system to perform this blending it must be able to search the tracks of the controller and see if tracks exist with the same group name. It is in this member that we store a string containing the group name of the set definition assigned to this track. When the CActionDefinition::ApplyAction method assigns the animation sets of an action to the tracks on the mixer, it will also store the group name associated with that set definition within the action in this member.

TCHAR strSetName[127]

When the action system assigns an animation set to a mixer track it will store the name of the animation set in this member. Although this information could be fetched by querying the animation set interface assigned to the track, it is more convenient to store it here. If the application wishes to use the CActor::GetActionStatus method to get a pointer to the

CActionStatus object, it will be able to easily examine the TrackActionStatus structures for each track to determine which animation sets are currently being used by the controller and the specified action.

const CActionDefinition * pActionDefinition
This member is set when the animation set is assigned to the mixer track. It contains a pointer to the action that set this track. This is useful to have because we know that due to the blending process, not all the animation sets currently set on the mixer will have been set by the currently applied action. If we imagine a case where we have three actions that each use three different animation sets which are configured to blend over a period of 20 seconds, and we apply these three actions over a period of 5 seconds, at 10 seconds (for example) there will actually be 9 animation sets currently in use. The first three will have been set by the first action but will still be in the process of fading out. The second three animation sets will have been set by the action that was applied second and these too will still be in the process of fading out. The final three sets will have been set by the currently applied action. If each track stores a pointer to the action that was responsible for setting its animation set, this may come in handy if the application needs to know this information.

ULONG nSetDefinition
This stores the index of the set definition. For example, if the animation set was referenced by the second set definition in the action (inside the .ACT file) this would be the index that was stored here. Once again, this is really for the purpose of providing as much information as we can to the application in the event that it needs it for some reason.

CActionDefinition *m_pCurrentAction
This member contains a pointer to the currently set action. This is set and retrieved by the CActionStatus::SetCurrentAction and CActionStatus::GetCurrentAction.

We saw the SetCurrentAction method of this object being called from the CActor::ApplyAction method discussed recently. The function first applied the action and then informed the CActionStatus object of the new action that was set using this method. We also saw how the CActor::GetCurrentAction method simply wraps the call to CActionStatus::GetCurrentAction. These Get and Set functions simply set and return the value of this pointer.

ULONG m_nRefCount
Contains the current reference count of the object.

Let us now discuss the methods of the CActionStatus structure. For the most part these will be simple member access functions. We will not cover the AddRef and Release methods here since at this point in the series you should be comfortable with what these methods look like.

CActionStatus:: CActionStatus

The constructor initializes all member variables to NULL or zero, except for the reference count of the object. Since we are in the constructor, the object is being created, so it must have a reference count of 1 when this function returns.

```
CActionStatus::CActionStatus( )
{
    // Clear required variables
    m_pCurrentAction     = NULL;
    m_pTrackActionStatus = NULL;
    m_nMaxTrackCount     = 0;

    // ***********************************
    // *** VERY IMPORTANT
    // ***********************************
    // Set our initial reference count to 1.
    m_nRefCount = 1;
}
```

CActionStatus::~CActionStatus

The destructor releases the TrackActionStatus array (if it exists) and sets the three other members to NULL or zero.

```
CActionStatus::~CActionStatus( )
{
    // Delete any flat arrays
    if ( m_pTrackActionStatus ) delete []m_pTrackActionStatus;

    // Clear required variables.
    m_pCurrentAction     = NULL;
    m_pTrackActionStatus = NULL;
    m_nMaxTrackCount     = 0;
}
```

CActionStatus::SetMaxTrackCount

This function resizes the TrackActionStatus array to the size specified by the Count parameter. It is called by the actor whenever the limits of the controller change. There should always be a 1:1 mapping between tracks on the mixer and TrackActionStatus structures in the internal array.

The function first tests the value of the Count parameter. If it is set to zero then it means the caller wishes to flush the per track data currently stored in the CActionStatus object. When this is the case, we delete the TrackActionStatus array and set the m_nMaxTrackCount member variable to zero.

```
HRESULT CActionStatus::SetMaxTrackCount( USHORT Count )
{
```

```
    TrackActionStatus * pBuffer = NULL;

    // If we're clearing out.
    if ( Count == 0 )
    {
        // Just release and return
        if ( m_pTrackActionStatus ) delete []m_pTrackActionStatus;
        m_pTrackActionStatus = NULL;
        m_nMaxTrackCount     = 0;

        // Success
        return S_OK;

    } // End if count == 0
```

In the next section of code, we allocate a new array of TrackActionStatus structures. Its size will equal the size requested by the Count parameter. We then initialize this array to zero.

```
    // Allocate enough room for the specified tracks
    pBuffer = new TrackActionStatus[ Count ];
    if ( !Count ) return E_OUTOFMEMORY;

    // Clear the buffer
    memset( pBuffer, 0, Count * sizeof(TrackActionStatus) );
```

If this CActionStatus object already has data in its TrackActionStatus array then this must be copied into the new array. This allows us to change the number of tracks the CActionStatus object can handle without losing any data that might already exist. This is obviously very important if the resize is performed while the action system is in use. After all, the application can call the CActor::SetActorLimits method at any time (which will cause this function to be called and passed the new track limit of the controller).

As you can see in the following code, if the current track count of the object is larger than zero and its TrackActionStatus member is not NULL then there must be data already and we should copy it over.

```
    // Was there any previous data?
    if ( m_nMaxTrackCount > 0 && m_pTrackActionStatus )
    {
        // Copy over as much data as we can
        memcpy( pBuffer,
                m_pTrackActionStatus,
                min(m_nMaxTrackCount, Count) * sizeof(TrackActionStatus) );

    } // End if existing data
```

Now that we have our new array and have the previous track data copied into it, we can delete our original array.

```
    // Release previous buffer
    if ( m_pTrackActionStatus ) delete []m_pTrackActionStatus;
```

Now we can assign the m_pActionStatus member to point at our new array instead. We also set the m_nMaxTrackCount member to the new track count of the controller that was passed in. The value in m_nMaxTrackCount should always reflect the number of elements in the m_pActionStatus array.

```
    // Store buffer, and count
    m_pTrackActionStatus = pBuffer;
    m_nMaxTrackCount     = Count;

    // Success!
    return S_OK;
}
```

CActionStatus::SetCurrentAction

This simple function assigns the passed CActionDefinition pointer as the current action. We saw this function being called earlier from CActor::ApplyAction. It first called the CActionDefinition::ApplyAction method to set up the controller for the action being applied and then it called this function to notify the CActionStatus object that the current action has changed.

There is also a 'get' version of this function that is inline; it returns the current action.

```
void CActionStatus::SetCurrentAction( CActionDefinition * pActionDefinition )
{
    // Just store the current action
    m_pCurrentAction = pActionDefinition;
}
```

So there is nothing very complicated about the CActionStatus object. It basically just provides the actor with a means to record the current state of the action system. It stores the current action that has been applied and information about the status of each track on the mixer.

As mentioned earlier, all of the hard work is done inside the CActionDefinition object. This class has two main functions that are called by the CActor. The CActionDefinition::LoadAction method was called from CActor::LoadActionDefinitions once for each action. It has the task of loading the information for the specified action from the .ACT file. The second important function is the CActionDefinition::ApplyAction function which is called by CActor::ApplyAction to actually perform the task of setting up the controller's tracks for the new action. This function is effectively the entire runtime action system. It sets up the transitions between sets that need to be faded in and out, it sets sequencer events to control those fade outs, and a host of other things. Let us now discuss the final component of our action system.

The CActionDefinition Class

This new class is designed for the storage and application of a single action. When the ACT file is loaded, each individual action (Walk, Run, etc.) is stored within a separate instance of a CActionDefinition object, inside the CActor member array. In addition, the member functions belonging to this class are used directly whenever an action is to be applied to the animation controller.

The class declaration is shown below and is contained in CActor.h. It may seem on first glance that this class is rather large and complex but do not be misled by the two enumerations and one new structure (used by the class) defined in its namespace. Actually, the class has only four member variables and five methods (not including the constructor and destructor) and all but two of those are simple access functions.

```
class CActionDefinition
{
public:

    // Public Enumerators for This Class.
    enum ActBlendMode                       // Flags for SetDefinition::BlendMode
    {
        Off          = 0,
        MixFadeGroup = 1
    };

    enum ActTimeMode                        // Flags for SetDefinition::TimeMode
    {
        Begin              = 0,
        MatchGroup         = 1,
        MatchSpecifiedGroup = 2
    };

    // Public Structures for This Class.

    struct SetDefinition
    {
        TCHAR        strSetName[128];       // The name of the animation set
        TCHAR        strGroupName[128];     // The 'group' to which this belongs.
        float        fWeight;               // Anim Set strength
        float        fSpeed;                // The speed of the set itself
        ActBlendMode BlendMode;             // Blending to use when set is applied
        ActTimeMode  TimeMode;              // Timing Source when set is applied
        float        fMixLength;            // Length of transition is blend
                                            // enabled
        TCHAR        strTimeGroup[128];     // If the time mode specified that we
                                            // should match the timing to a
                                            // specific alternate group, that
                                            // group name is stored here.

        // Internal Temporary Vars
        bool         bNewTrack;             // Assign this set to a new track
        double       fNewTime;              // Use this time for the new track.
    };
```

```
    // Constructors & Destructors for This Class.

    CActionDefinition( );
    virtual ~CActionDefinition( );


    // Public Functions for This Class.

    HRESULT         LoadAction              ( ULONG nActionIndex,
                                              LPCTSTR ActFileName );

    HRESULT         ApplyAction             ( CActor * pActor );

    // Accessor functions
    LPCTSTR     GetName                 ( ) const  { return m_strName; }
    ULONG       GetSetDefinitionCount   ( ) const{ return m_nSetDefinitionCount; }

    SetDefinition * GetSetDefinition( ULONG Index ) const
                                    { return ( Index < m_nSetDefinitionCount )?
                                      &m_pSetDefinitions[ Index ] :
                                      NULL; }


private:

    // Private Variables for This Class.

    TCHAR           m_strName[128];         // The name of this action definition
    USHORT          m_nSetDefinitionCount;  // The number of set definitions
    SetDefinition * m_pSetDefinitions;      // Array of set definition structures.
};
```

We will start by discussing the member variables of this class and then we will finally wrap up our coverage of the Action Animation System by examining the code to the member functions.

TCHAR m_strName[128]

This member will hold the name of the action as read in from the .ACT file. In the example .ACT file we looked at earlier, the first action was called ‘Walk_At_Ease’. This is the name that would be stored in this member for that action. This value will be set when the CActionDefinition::LoadAction method is called and the name of the current action being loaded is extracted from the file.

Notice in the above code that there is an inline method called GetName which will return the name of the action. We saw this function being used in the CActor::ApplyAction method earlier. It searched through all of the actions using the GetName method to test against the name of the requested action passed by the application. If a match was found, this was assumed to be the action that the application would like to apply and its ApplyAction method was called. We will look at the CActionDefinition::ApplyAction method in a moment.

USHORT m_nSetDefinitionCount
This member contains the number of SetDefinitions that exist for this action. In our example .ACT file, each action contained three set definitions. Recall that a set definition is really just the name of an animation set that is used to comprise the action and any associated properties that should be associated with that set (such as should it transition in or not). Every action can have a different number of set definitions, so one action may involve setting a single animation set while another action may set many.

This member contains the number of animation sets that will need to be set on the mixer when this action is applied. This also describes the number of elements in the m_pSetDefinitions array (discussed below) which is an array that contains the information for each set definition.

SetDefinition m_pSetDefinitions
As discussed above, each action is usually comprised of several set definitions. Each set definition describes the name and properties of an animation set that should be assigned to the mixer to help achieve the action when it is applied. The number of elements in this array will be equal to the number of set definitions defined in the .ACT file for the current action. This array will be allocated when the action is loaded.

The name and properties of each animation set used by this action are stored in this array as a SetDefinition structure. Let us now describe the members of the SetDefinition structures that comprise this array.

> **TCHAR strSetName[128]**
> This contains the name of the animation set that this set definition is associated with. You will recall that when you defined a set definition in the .ACT file you had to specify the first property as the name of the animation set you would like these properties to be linked to. The animation set you name here will be one (of potentially many) that will be assigned to the animation mixer when this action is applied.
>
> **TCHAR strGroupName[128]**
> Here we store the name of the group this set definition is assigned to. This is pretty important when blending is being used since we will often want an animation set to transition from another animation set with the same group name when the action is applied. As discussed earlier, you should never have more than one set definition within a single action with the same group name.
>
> **float fWeight**
> The initial weight that you would like the animations set to have. For an animation set that is configured to fade-in, this is the target weight that the animation set's track will reach at the end of the fade-in process. For non fade-in sets, the track weight will be assigned this value the moment the set is activated.
>
> **float fSpeed**
> The speed at which the animation set should play. You will usually set this to 1.0 so that the animation plays at the default speed intended by the asset creator.

ActBlendMode BlendMode

This set definition property is used to determine whether the animation set should be played immediately at full weight (disabling any animation set currently playing from the same group) or whether it should slowly transition in. It will be set to one of the following members of the ActBlendMode enumerated type. This enumeration is defined within the CActionDefinition namespace in CActor.h. As you can see, it mirrors the two values we could use in the .ACT file to specify the blend mode.

```
enum ActBlendMode
{
        Off             = 0,
        MixFadeGroup    = 1
};
```

Off

When set to Off, no blending occurs. When the action is applied, any previous animation sets (which have not already been put into a transition out state) from the same group will be immediately halted and replaced with this new set.

MixFadeGroup

When the action is applied, any animation set currently playing from the previous action with a matching group name will be faded out over a period of time and the strength of this set will be faded in over that same period of time. This causes a smooth transition between animation sets in the same groups; animation sets that animate the same portions of the character for example.

ActTimeMode TimeMode

The TimeMode property of a set definition is something else we discussed when covering the .ACT file specification. The CActionDefinition::LoadAction will set the TimeMode property of each of its set definitions to one of three members of the ActTimeMode enumerated type (depending on which of these values is specified in the .ACT file).

```
enum ActTimeMode
{
        Begin               = 0,
        MatchGroup          = 1,
        MatchSpecifiedGroup = 2
};
```

Begin

The animation set will start from the beginning when played. That is, the track timer to which it is assigned will be reset. This is used mainly if the animation set you are playing does not have to be synchronized either to another animation set that is currently animating another part of the actor or if you do not need this animation set to smoothly take its periodic position from an animation set in the same group that is already playing from a previous action.

MatchGroup

In this mode, when the animation set is assigned to the track, its track time will be copied from the track of an animation set in the same group that was being used by the previous action.

A good example of a need for this mode would be if two actions contain the exact same animation set for the 'Legs' group. Both actions may have different animation sets for the torso but may both use the 'Walk' animation set for the legs. If action 1 was currently being played and the player was halfway through a stride, we would not want the animation set working the legs to have its periodic position snapped back to zero when the second action was applied. Instead, we would want the animation set working on the legs in the second action to start at a periodic position that continues the stride of the character and takes over from where the 'Legs' animation set in the previous action left off.

MatchSpecifiedGroup

This mode is useful for synchronizing the different animation sets together. In out example .ACT file, the torso animation for the 'Walk_At_Ease' action uses this mode so that it can take its timing from the 'Legs' group. That way the legs and arms are synchronized as the character walks.

float fMixLength

If the blend mode of this SetDefinition has been set to MixFadeGroup, then this value will contain the number of seconds this animation set will take to transition to full strength and how long any animation set from a previous action in the same group will take to have its strength completely faded out.

If the BlendMode is set to 'Off', this property will not be used.

TCHAR strTimeGroup[128]

If the TimeMode of the SetDefinition has been set to MatchSpecifiedGroup then this member will contain the name of the group that this animation set will take its track time from when first applied to the mixer. For example, a SetDefinition which has been assigned to the 'Arms' group could have its TimeMode set to take its initial starting position from the 'Legs' group animation set that is currently playing.

If TimeMode is not set to MatchSpecifiedGroup, this property is not used.

bool bNewTrack

double fNewTime

These final two members of the SetDefinition structure are used for temporary data storage during the processing of an action inside the CActionDefinition::ApplyAction method. We will see them being used later.

We have now discussed all the member variables and structures used by CActionDefinition. Let us now look at its member functions.

CActionDefinition::CActionDefinition

The constructor places the terminating null character in the first position of the name string (empty string) and sets the definition count to zero. It also sets the definition set array pointer to NULL. These variables will not be populated with meaningful values until the CActionDefinition::LoadAction method is called to populate this object.

```
CActionDefinition::CActionDefinition( )
{
    // Clear required variables
    m_strName[0]            = _T('\0');
    m_nSetDefinitionCount   = 0;
    m_pSetDefinitions       = NULL;
}
```

CActionDefinition::~CActionDefinition

The destructor quite expectedly releases the set definitions array and sets its pointer to NULL and restores all values to their default state.

```
CActionDefinition::~CActionDefinition( )
{
    // Delete any flat arrays
    if ( m_pSetDefinitions ) delete []m_pSetDefinitions;

    // Clear required variables.
    m_strName[0]            = _T('\0');
    m_nSetDefinitionCount   = 0;
    m_pSetDefinitions       = NULL;
}
```

CActionDefinition::LoadAction

This method is called by the CActor::LoadActionDefinitions method. You will recall when we covered that function earlier that it read the number of actions in the .ACT file and then allocated the actor's array of CActionDefinition objects. It then looped through each action and called its LoadAction function to populate it with the action data from the .ACT file.

LoadAction is a little long and rather uneventful. It just uses the Win32 .ini file reading functions to read the various properties of the action from the file and store that data in its internal members.

The function is passed two parameters by the CActor::LoadActionDefinitions function. The first is the index that will be appended to the word 'Action' to describe the Action block in the .ACT file that contains the data for this action definition. The second parameter is the name (and path) of the .ACT file that we are currently in the process of fetching the action information from.

The first thing this function does is build a string (using the _stprintf function) containing the name of the data block in the .ACT file we wish to access. Once we have this string constructed and stored in the local variable (strSectionName) we flush any data that may have previously existed in this action definition object. We release its SetDefinitions array if it exists and make sure that its definition count is set to zero at the start of the loading process.

```
HRESULT CActionDefinition::LoadAction( ULONG nActionIndex, LPCTSTR ActFileName )
{
    TCHAR strSectionName[128];
    TCHAR strKeyName[128];
    TCHAR strBuffer[128];
    ULONG i;

    // First build the action section name
    _stprintf( strSectionName, _T("Action%i"), nActionIndex );

    // Remove any previous set definitions
    if ( m_pSetDefinitions ) delete []m_pSetDefinitions;
    m_pSetDefinitions     = NULL;
    m_nSetDefinitionCount = 0;
```

The first property we fetch is the DefinitionCount. Every action in the .ACT should have a definition count property which describes how many set definitions comprise the action. This also tells us how large we should allocate the m_pSetDefinitions array for this action.

As before, because the value we wish to read is an integer value, we use the Win32 function designed for the task of reading integer values from an .ini file (GetPrivateProfileInt).

```
    // Retrieve the set definition count
    m_nSetDefinitionCount = ::GetPrivateProfileInt(  strSectionName,
                                                     _T("DefinitionCount"),
                                                     0,
                                                     ActFileName );

    // If there are no sets, we were most likely unable to find the action
    if ( m_nSetDefinitionCount == 0 ) return D3DERR_NOTFOUND;

    // Allocate the setdefinition array
    m_pSetDefinitions = new SetDefinition[ m_nSetDefinitionCount ];
    if ( !m_pSetDefinitions ) return E_OUTOFMEMORY;
```

The first parameter to GetPrivateProfileInt is the string we built at the start of the function. It contains the name of the action block in the .ACT file we wish to read data from. As the second parameter we pass a string describing the property we wish to read within this action block. In this call we want to fetch the value of the DefinitionCount property. You must type property names exactly as you have defined them in the .ACT file. As the third parameter we always pass in a default value that will be used if the property cannot be found in the file or in the specified block/section. In this case, if the Definition Count property for an action is missing, this is considered a corrupt action description and we assign the value of zero. This will cause the function to exit on the very next line and the action will not be loaded. As the final parameter to this function we pass the name of the .ACT file we wish to read.

On function return, we should have the set definition count in the member variable m_nSetDefinitionCount and if this value is zero, we return from the function and do not load this action. Otherwise, we use this value to allocate an array of SetDefinition structures of the correct size.

We will now fetch the name of the action. Since the name of the action is a string and not an integer value we will use the Win32 GetPrivateProfileString function to fetch this data. The function works in an almost identical way but takes extra parameters. The first parameter is the block name in the .ACT file we wish to pull the values from. This is the current Action block being read. As the second parameter we pass the name of the property within this block that we wish to read. In this next call, we wish to read the value of the Name property for this action. As the third parameter we pass in the string that should be returned if the property is not found; in this case we simply specify an empty string. As the fourth parameter we pass in the string variable that will contain the value on function return. We wish to store the name of the action in the m_strName member variable. The fifth parameter is the number of character we would like to read. This is to make sure that we do not overflow the memory bounds of our string. We read the first 127 characters, since an action name is not likely to ever be longer than this and our m_strName member variable only has space for 128 characters (i.e., 127 characters plus the terminating NULL).

```
    // Retrieve the name of the action
    ::GetPrivateProfileString( strSectionName,
                               _T("Name"),
                               _T(""),
                               m_strName,
                               127,
                               ActFileName );
```

At this point we have the name of the action stored, the number of set definitions it should contain and we have allocated the SetDefinitions array. Let us now loop through each element in that array and populate each set definition with data from the file.

In the first section of the loop (shown below) we get a pointer to the current element in the SetDefinitions array. First we read the name of the animation set and the name of the group associated with this definition and store them in the definition set's strSetName and strGroupName members.

```
    // Loop through each set defined in the file
    for ( i = 0; i < m_nSetDefinitionCount; ++i )
    {
        // Retrieve a pointer to the definition for easy access
        SetDefinition * pDefinition = &m_pSetDefinitions[i];

        // Get the set name
        _stprintf( strKeyName, _T("Definition[%i].SetName"), i );
        ::GetPrivateProfileString( strSectionName,
                                   strKeyName,
                                   _T(""),
                                   pDefinition->strSetName,
                                   127,
                                   ActFileName );

        // Get the group name
```

```
        _stprintf( strKeyName, _T("Definition[%i].GroupName"), i );
        ::GetPrivateProfileString(  strSectionName,
                                    strKeyName,
                                    _T(""),
                                    pDefinition->strGroupName,
                                    127,
                                    ActFileName );
```

Notice that before we access each property, we build a string containing the name of the property we wish to fetch. For example, if we are fetching the animation set name of the third set definition, the name of this property inside the action's data block would be 'Definition3.SetName'. Similarly, the group name would be assigned to the property 'Definition3.GroupName'. Open up the .ACT file and examine it for yourself so you can follow along with this function.

In the next section of code we need to determine if this set definition is set to blend or not. The blend mode of this definition will be set to either 'Off' or 'MixFadeGroup'. Notice in the following function call to GetPrivateProfileString that a default string of 'Off' is returned if no matching property has been specified in the .ACT file. This means if no blend mode is specified, it is equivalent to specifying no blending.

```
        // Get the blend mode flags
        _stprintf( strKeyName, _T("Definition[%i].BlendMode"), i );
        ::GetPrivateProfileString(  strSectionName,
                                    strKeyName,
                                    _T("Off"),
                                    strBuffer,
                                    127,
                                    ActFileName );


        if ( _tcsicmp( strBuffer, _T("MixFadeGroup") ) == 0 )
            pDefinition->BlendMode = ActBlendMode::MixFadeGroup;
        else
            pDefinition->BlendMode = ActBlendMode::Off;
```

Once we have fetched the blend mode string we test its value and set the definition's BlendMode member to the correct enumeration.

We will now read the TimeMode of the set definition. It can be either 'Begin', 'MatchGroup' or MatchSpecifiedGroup. A default string of 'Begin' will be returned if no TimeMode property exists in the .ACT file for the current set definition being read. In other words, if no time mode property exists in the .ACT file for a given set definition, by default time the animation set will be started from the beginning whenever the action is applied.

```
        // Get the time mode flags
        _stprintf( strKeyName, _T("Definition[%i].TimeMode"), i );
        ::GetPrivateProfileString(  strSectionName,
                                    strKeyName,
                                    _T("Begin"),
                                    strBuffer,
                                    127,
```

```
                                    ActFileName );

        if ( _tcsicmp( strBuffer, _T("MatchGroup") ) == 0 )
            pDefinition->TimeMode = ActTimeMode::MatchGroup;
        else
            if ( _tcsicmp( strBuffer, _T("MatchSpecifiedGroup") ) == 0 )
            pDefinition->TimeMode = ActTimeMode::MatchSpecifiedGroup;
        else
            pDefinition->TimeMode = ActTimeMode::Begin;
```

After we have retrieved the string from the file we test its contents and set the definition's TimeMode member to the correct member of the ActTimeMode enumeration.

In the next section we retrieve the Weight and Speed properties for this set definition. Since these are floating point values and Win32 exposes no GetPrivateProfileFloat function, we will have to read the values as strings and then convert the strings into floats using the _stscanf function.

You can see in the following code that after we extract the weight value (with a default weight of 1.0 if the property is not found in the file), we have ourselves a floating point value stored in the string strBuffer. We then use the _stscanf function to fetch the float from that string and store it in the fWeight member for the current definition. The first parameter to the _stscanf is the string we wish to extract the values from and the second parameter is a format string that informs the function how many values we wish to extract and what their types are. Using the '%g' string tells the function that we wish to extract one floating point value from the beginning of the string. Since we are only extracting one value, there is only one variable specified after this parameter -- the address of the variable we would like this value copied into. We perform the same procedure to extract the Speed property as well.

```
        // Retrieve and scan the weight float var
        _stprintf( strKeyName, _T("Definition[%i].Weight"), i );
        ::GetPrivateProfileString(  strSectionName,
                                    strKeyName,
                                    _T("1.0"),
                                    strBuffer,
                                    127,
                                    ActFileName );

        _stscanf( strBuffer, _T("%g"), &pDefinition->fWeight );

        // Retrieve and scan the speed float var
        _stprintf( strKeyName, _T("Definition[%i].Speed"), i );
        ::GetPrivateProfileString(  strSectionName,
                                    strKeyName,
                                    _T("1.0"),
                                    strBuffer,
                                    127,
                                    ActFileName );

        _stscanf( strBuffer, _T("%g"), &pDefinition->fSpeed );
```

At this point we have read in the standard properties that are applicable in all circumstances, but we now have to read two optional ones. For example, we know that if the BlendMode of this set definition is set

to MixFadeGroup then there will also be a MixLength property specified for this set which describes how long the transition should take to perform. This is a float value that specifies this time in seconds so once again, we fetch the value as a string and scan it into the set definition's fMixLength member.

```
        // If we're blending, there will be an additional parameter
        if ( pDefinition->BlendMode == ActBlendMode::MixFadeGroup )
        {
            // Retrieve and scan the various floats
            _stprintf( strKeyName, _T("Definition[%i].MixLength"), i );
            ::GetPrivateProfileString(    strSectionName,
                                          strKeyName,
                                          _T("0.0"),
                                          strBuffer,
                                          127,
                                          ActFileName );

            _stscanf( strBuffer, _T("%g"), &pDefinition->fMixLength );

            // If we returned a zero value, turn off mix fading of the group
            if ( pDefinition->fMixLength == 0.0f )
                  pDefinition->BlendMode = ActBlendMode::Off;

        } // End if fading
```

Notice that if no MixLength property is found, a default of zero is returned. This causes the definition's BlendMode member to be set to ActBlendMode::Off (i.e., blending is disabled).

The last value we read in for the current definition is the TimeGroup property. It will only be specified in the .ACT file if the definition has had its TimeMode set to MatchSpecifiedGroup. Recall that this means when the animation set is assigned to the mixer, it will have its track time set to the position of an existing track in the specified group. This property contains the name of that group.

```
        // If we're timing against a specified group, get additional parameter
        if ( pDefinition->TimeMode == ActTimeMode::MatchSpecifiedGroup )
        {
            // Retrieve and scan the various floats
            _stprintf( strKeyName, _T("Definition[%i].TimeGroup"), i );
            ::GetPrivateProfileString(    strSectionName,
                                          strKeyName,
                                          pDefinition->strGroupName,
                                          pDefinition->strTimeGroup,
                                          127,
                                          ActFileName );

        } // End if matching against alternate group

    } // Next new set definition

    // Success!!
    return S_OK;
}
```

That is the end of the loop and the end of the function. This loop is executed for each definition in the current action. At the end, the CActionDefinition object will have a fully populated SetDefinition array

where each element can be used by the ApplyAction method to setup the animation sets correctly when the action is applied.

CActionDefinition::ApplyAction

This is the function that provides the core processing of our Action Animation System. Not only does it set the animation sets for the action on the animation mixer, it also performs fade outs of the animation sets from a previous action. Additionally, it keeps the actor's CActionStatus object up to date so that we always know which tracks are in use, which ones are transitioning out, and which ones are free for use.

The function can appear intimidating due to its sheer size. However once you take a closer look, you should see that it is really not so difficult. The code is really repeating the same steps for both the blending and non-blending cases, so it should be pretty easy to grasp once you understand the main logic.

This function is called by the CActor::ApplyAction function after it has found the CActionDefinition object with a name that matches the one requested by the application. Once the correct action is found, its ApplyAction method is called to configure the animation mixer. Let us look at that code now.

The first thing this function does is get a pointer to the actor's CActionStatus object. As this contains the current state of each track (is it in use or not for example) we will definitely need this information.

```
HRESULT CActionDefinition::ApplyAction( CActor * pActor )
{
    SetDefinition                    * pDefinition;
    CActionStatus                    * pStatus;
    CActionStatus::TrackActionStatus * pTrackStatus;
    ULONG                              i, j, nCount;
    D3DXTRACK_DESC                     TrackDesc;

    // Retrieve the actors action status
    pStatus = pActor->GetActionStatus();
    if ( !pStatus ) return D3DERR_INVALIDCALL;
```

This step is very important for finding tracks that have fully transitioned out and updating the relevant track properties in the CActionStatus object to reflect that this track is now free.

We said earlier when discussing how this system will work, that if a given set definition is configured to blend, then its weight will transition in over a period of time and a previous animation set that was playing from the same group will transition out over the same period. This produces a smooth blend from the old set to the new one. The fading out of the set is done by setting an event on the previous animation set's track that will reduce its weight to zero over this period. We then set another event on that track that will be triggered when the weight reaches zero which will disable the track. If a track has been disabled since we last applied any actions, we need to know about it so that we can update the track status in our CActionDefinition object. If we do not do this, our system will think the track is still in use and we will find ourselves running out of tracks in a very short time as actions are repeatedly applied.

The next section of code performs this synchronization process between the tracks on the animation mixer and the TrackActionStatus structures in our CActionStatus object. Let us first look at the code and then we will discuss it.

```
    // Mark disabled tracks as free for use.
    nCount = pActor->GetMaxNumTracks();

    for ( j = 0; j < nCount; ++j )
    {
        // Retrieve action status information for this track.
        pTrackStatus = pStatus->GetTrackStatus( j );

        // Get the current track description.
        pActor->GetTrackDesc( j, &TrackDesc );

        // Flag as unused if the track was disabled
        if ( TrackDesc.Enable == FALSE )
        {
            pTrackStatus->bInUse            = false;
            pTrackStatus->bTransitioningOut = false;

        } // End if track is disabled
    } // Next track
```

We first fetch the number of tracks on the animation mixer and set up a loop to process each track. For each track, we get a pointer to the associated TrackActionStatus structure in our CActionStatus object and also call the actor's GetTrackDesc method to fetch a D3DXTRACK_DESC containing the properties of the actual mixer track. Then we test to see if the track on the mixer has been disabled. If it has, the Enable member of the D3DXTRACK_DESC structure will be set to false. As you can see, if it is, it means that this track is no longer being used by our action system and we should set the bInUse and bTransitioningOut booleans of the corresponding track on our CActionStatus object to false. We will now recognize that this track is available for use by our system. All we are really doing is performing garbage collection and making sure we recycle any tracks that were being used for transitioning out but have since been disabled by a sequencer event that occurred when the weight of the track reached zero.

At this point, our CActionStatus object correctly reflects the current state of the tracks with respect to availability. Now it is time to loop through each of the set definitions in this action and figure out which track the definition should be assigned to, and what its starting position and transition status should be. With the exception of a small piece of code at the bottom, most of the logic in this function is contained inside this loop. That is, all the code you are about to see is carried out for each set definition in the action. Remember, a set definition represents an animation set and its associated properties.

```
    // Loop through each of our set definitions
    for ( i = 0; i < m_nSetDefinitionCount; ++i )
    {
        // Retrieve the definition pointer for easy access
        pDefinition = &m_pSetDefinitions[i];

        // Reset sensitive variables
        pDefinition->bNewTrack = false;
        pDefinition->fNewTime  = 0.0f;
```

The first thing we do at the head of this loop (above) is get a pointer to the current set definition we are processing. From this point on, pDefinition will be the pointer we use for ease of access.

We also reset the bNewTrack and fNewTime variables to false and zero respectively. You may recall when we discussed the members of the SetDefinition structure that these two members were described as temporary storage variables. They were not populated by the loading process with values from the .ACT file but will be used in this function to record whether this definition needs to be assigned to a new track and the starting position the animation set should play from. We set these to default values for now because we do not yet know if a new track will be required or if this animation set is currently assigned to a track already and can be recovered. We have also not yet calculated what the position of the track should be when the set is assigned. We will need to determine this in a moment based on the TimeMode that has been set for this definition. If the TimeMode of this set has been set to 'Begin' then zero will be the track position we wish it to start from.

At the end of this loop, in each definition we will have stored whether or not a new track is needed for it and how the track position should be set. We can then do a final pass through the tracks on the mixer and start our assignments. This will make a lot more sense as we progress; for now just know that we are just trying to determine the track position for each set and whether a new track is needed.

If the TimeMode for this set definition is set to 'Begin' then we have already stored the correct track position in the definition's fNewTime member. However, if one of the other time modes have been activated, we will try to find a track that is already active which has an animation set assigned to it in the same group (or the specified group) and will inherit its time to ensure a smooth takeover. The entire next section of code is executed only if this set definition's time mode is set to either MatchGroup or MatchSpecifiedGroup.

```
        // Get timing data from actor if required
        if ( pDefinition->TimeMode != ActTimeMode::Begin )
        {
            // Choose which group name to pull timing from.
            TCHAR * pGroupName = pDefinition->strGroupName;

            if ( pDefinition->TimeMode == ActTimeMode::MatchSpecifiedGroup )
                 pGroupName = pDefinition->strTimeGroup;
```

In the above code we first test to see which of the two timing match modes this set definition has been configured for. If the TimeMode being used is set to MatchGroup then the local variable pGroupName will point to the name of this definition's group. If MatchSpecifiedGroup is being used, then this same pointer is instead assigned to the value stored in the definition's str_TimeGroup member. Either way, this means at the end of this code, the local variable pGroupName contains the name of the group we wish to inherit the time from (if possible). We will now need to test each track currently assigned to the mixer and inherit its time if it belongs to the same group. After examining all this function code and the steps involved in transitioning between two actions, you will understand that there will only ever be one currently active track on the mixer with the same group name. This does not include tracks which are transitioning out since they are not considered to be currently active. As such, they are left to their own devices to fade out. Therefore we are now going to search for this track and inherit its track position.

In the next section of code we set up a loop to loop through each track on the mixer. For each track, we fetch the corresponding TrackActionStatus structure from our CActionStatus object so we can examine if the track is currently in use or not. If the track is not in use or it is in the process of transitioning out then this track is essentially not currently active and we have no interest in inheriting its track position. So we skip it. If however, the track is currently in use by the action system and has a definition assigned to it with a group name that matches the one we wish to take our timing from, we will fetch the position of that track (using our CActor::GetTrackDesc function) and will assign this position to the start time stored in the current set definition we are processing. This entire process is shown below.

```
            // Find the track in this group not currently transitioning out
            nCount = pActor->GetMaxNumTracks();
            for ( j = 0; j < nCount; ++j )
            {
                // Retrieve action status information for this track.
                pTrackStatus = pStatus->GetTrackStatus( j );

                // Skip if this track is not in
                if ( pTrackStatus->bInUse == false ||
                     pTrackStatus->bTransitioningOut == true ) continue;

                // Does this belong to this group?
                if ( _tcsicmp( pTrackStatus->strGroupName, pGroupName ) == 0 )
                {
                    // Get the current track description.
                    pActor->GetTrackDesc( j, &TrackDesc );

                    // Store the position and break
                    pDefinition->fNewTime = TrackDesc.Position;
                    break;

                } // End if matching group

            } // Next Track

        } // End if need timing data
```

At this point, we have processed the time modes and our fNewTime member stores the position that its track should be set to start at when we do eventually assign it to a track at the tail of the function. The timing processing is now all done.

In the next phase we have to figure out whether this animation set needs to be blended in and a currently active set from the same group faded out. Unfortunately, things are not quite as black and white as it might seem. If the animation set for this set definition that we wish to apply matches the name of an animation set already assigned to the mixer by a previous action and the group names match, it would make little sense to have the two exact sets, one fading out and one fading in, taking up two valuable tracks on the mixer. Therefore, this is actually going to be a two step process. If mixing is being used and an exact animation set is currently in use, we will just recover it. We will just set it to fade back in, from whatever its current strength is to whatever is desired by the set definition we are currently processing. If an exact set and group combination does not exist, then a new track will be needed for this set definition. If an animation set is assigned to another track which is part of the same group, it will be faded out.

Of course, in order to do any of this we must first know whether the animation mixer currently has an animation set name/group combo assigned to one of its tracks which is exactly the same as the one contained in the set definition we are about to apply. The next section of code performs this search.

A loop is constructed to test each TrackActionStatus structure in the CActionStatus object. For each track we find which is flagged as currently in use we will see if its group name matches the group name of the set definition we wish to apply. If it does, then we know the groups match so we will continue on to see if the animation set names match. If they do, we have found an exact match and the set definition we wish to apply will not need a new track. It can simply take over the track of this identical combination. As soon as a match is found the loop breaks and the loop variable's value is assigned to the local variable nSetMatched. This will contain the index of track on the animation mixer that the set definition we are about to apply will recover.

```
            long nSetMatched = -1;

            // First search for an exact match for this group / set combination
            nCount = pActor->GetMaxNumTracks();
            for ( j = 0; j < nCount; ++j )
            {
                  // Retrieve action status information for this track.
                  pTrackStatus = pStatus->GetTrackStatus( j );

                  // Skip if this track is not in use
                  if ( pTrackStatus->bInUse == false ) continue;

                  // Does this belong to this group?
                  if ( _tcsicmp(    pTrackStatus->strGroupName,
                                    pDefinition->strGroupName     ) == 0 )
                  {
                        // Is it physically the same set?
                        if ( _tcsicmp( pTrackStatus->strSetName,
                                       pDefinition->strSetName ) == 0 )
                              nSetMatched = (signed)j;

                        if ( nSetMatched >= 0 ) break;

                  } // End if matching group

            } // Next Track
```

All we have determined at this point is a value for nSetMatched which will either contain the index of a track to be recovered or -1 if no set/group combination currently exists. The value of this variable is used to make decisions in the next two (very large) conditional code blocks. The first code block processes the case when the definition we are about to apply require blending and the second contains almost identical code that processes the non-blending case. Even in the non-blending case, we will still recover a track with a matching set / group instead of assigning it a new track. It would be pointless to turn off one track to enable another with the exact same animation set assigned. We may as well just keep the one we were using before. Note in the above code that even if the track is transitioning out, we will recover it and flag it as a match so that we can recover it and fade it back in later.

Let us start to look at the blending case one section at a time, as it is a rather large code block. Indeed, this code block is further divided into two conditional code blocks which are executed depending on whether we are assigning a new track or recovering a track for this definition.

```
        // Does this definition require mix blending?
        if ( pDefinition->BlendMode == ActBlendMode::MixFadeGroup )
        {
```

The next block of code is executed if, during the previous search, we found that a track on the mixer currently has the same animation set with the same group name as the definition we wish to apply.

```
            // Did we find an exact match for this group / set combination?
            if ( nSetMatched >= 0 )
            {
                // This is the same set as we're trying to blend in, first thing
                // is to kill all events currently queued for this track
                pActor->UnkeyAllTrackEvents( nSetMatched );

                // Clear status
                pTrackStatus = pStatus->GetTrackStatus( nSetMatched );
                pTrackStatus->bInUse           = true;
                pTrackStatus->bTransitioningOut = false;
                pTrackStatus->pActionDefinition = this;
                pTrackStatus->nSetDefinition    = i;
```

If we are going to take over this track with our new set definition, we must be prepared for the fact that this track may have been in the process of being transitioned out when a previous action was applied. If this is the case then it will have a sequencer event set for that track that is due to disable the track when the transitioning out is complete. We no longer wish this to happen since we are going to recover the track and start to fade it back in. Therefore, in the above code we un-key any events that were set for the track so that they are no longer executed (thus preventing disabling the track). We then update the corresponding track properties in the CActionStatus object to reflect that this track is now in use again and is not transitioning out. We also store the CActionStatus object responsible for setting up this track as well as the set definition index. This is helpful if the application wishes to query this information.

Because the track that we have recovered may have been transitioning out or may have had a different weight than the one in the new set definition we are applying, we will set a KeyTrackWeight event that will be triggered immediately (notice the pActor->GetTime function call in the parameter list). It will fade the current weight of the track either in or out to match the weight of the set definition we are processing. Notice that the duration of the fade is determined by the definition's fMixLength property (specified in the .ACT file).

```
                // Queue up a weight event, and bring us back to the blend weight
                // we requested
                pActor->KeyTrackWeight(   nSetMatched,
                                          pDefinition->fWeight,
                                          pActor->GetTime(),
                                          pDefinition->fMixLength,
                                          D3DXTRANSITION_EASEINEASEOUT );
```

At this point, we have set up the track to transition correctly into the weight for the current set definition, but we also have to set the position of the track. Just because we are recovering the track does not mean we want our new definition to inherit the current position property of the track. This depends on the time mode of the definition specified in the .ACT file. In the next section of code, we fetch the track description and set its position to the time we calculated earlier. We also set its speed to the speed value for this definition. Of course, if this definition is set to inherit the time from its group, then we will absolutely leave the track position alone since it is already correct.

```
                // Get the current track description.
                pActor->GetTrackDesc( nSetMatched, &TrackDesc );


                // We are recovering from an outbound transition, so we will
                // retain the timing from this set, rather than the one we chose
                // above, if they specified only 'MatchGroup' mode.
                if ( pDefinition->TimeMode != ActTimeMode::MatchGroup )
                     TrackDesc.Position = pDefinition->fNewTime;

                // Setup other track properties
                TrackDesc.Speed = pDefinition->fSpeed;

                // Set back to track
                pActor->SetTrackDesc( nSetMatched, &TrackDesc );

            } // End if found exact match
```

At this point we have set up the set definition and the mixer track to recover. The definition's bNewTrack member will still be set to false as we do not need to assign it a new track.

The above code concludes the code block that deals with the case when we have found a matching track/group combination for our set definition in the blending case. Just to recap, the above code essentially says the following: "If I am supposed to transition in and a track already has the correct set and group name applied to it, recover this track and fade it back in".

The section of code we will see is still inside the blending case code block but is only executed if the set definition we wish to blend in does not match an animation set name and group name already applied to a mixer track. When this is the case, we simply set the definition's bNewTrack member to true so we will know at the bottom of the function that we will need to assign this set definition to a new track on the mixer.

```
            else
            {
                // This definition is now due to be assigned to a new track
                pDefinition->bNewTrack = true;

            } // End if no exact match
```

The next section of code is executed in the blending code block regardless of whether a track was recovered or we need a new track assignment for this definition. It has the job of finding all tracks currently in use that are in the same group as the current definition and then fading them out. There

should only be one active track in the same group that is not transitioning out. We are essentially searching for this track so that we can start to fade it out.

The following code starts in a similar manner to the previous search code. It searches through all the tracks in the CActionStatus object and searches for one with the same group name that is in use and not in the process of transitioning out. If found, we un-key any events it may have scheduled and set a new event to execute immediately that will reduce its weight to zero or the period specified by the fMixLength property of the set definition we are about to apply. This will fade the old group set out over the same duration as our new set will be faded in.

```
            // Find all tracks in this group that are not currently
            // transitioning out, and fade them
            nCount = pActor->GetMaxNumTracks();
            for ( j = 0; j < nCount; ++j )
            {
                // Skip this track if it's our matched track
                if ( (signed)j == nSetMatched ) continue;

                // Retrieve action status information for this track.
                pTrackStatus = pStatus->GetTrackStatus( j );

                // Skip if this track is not in use or is transitioning out
                if ( pTrackStatus->bInUse == false ||
                     pTrackStatus->bTransitioningOut == true )
                     continue;

                // Does this belong to this group?
                if ( _tcsicmp( pTrackStatus->strGroupName,
                               pDefinition->strGroupName ) == 0 )
                {
                    // Kill any track events already setup
                    pActor->UnkeyAllTrackEvents( j );

                    // Queue up a weight event to fade out the track
                    pActor->KeyTrackWeight( j,
                                            0.0f,
                                            pActor->GetTime(),
                                            pDefinition->fMixLength,
                                            D3DXTRANSITION_EASEINEASEOUT );

                    pActor->KeyTrackEnable( j,
                                            FALSE,
                                            pActor->GetTime() +
                                                      pDefinition->fMixLength );

                    // Update status
                    pTrackStatus->bInUse            = true;
                    pTrackStatus->bTransitioningOut = true;

                } // End if matching group

            } // Next Track

        } // End if requires blending
```

And that is the end of the blending case. Notice in the above code that not only do we set an event to fade the contribution of the previous track out over a period of time, we also key another event for when that period of time has been exceeded that will disable the track on the animation mixer. So, if the set definition we are about to apply had an fMixLength value of 5 seconds, we would trigger an event immediately that would fade the old set out over a 5 second period. We also set a KeyTrackEnable event that will disable the track in 5 seconds time when the fade out is complete. We saw at the start of the code that the next time an action is applied, a loop is performed through all the tracks to find ones that are disabled (and we updated the CActionStatus object accordingly). Finally, at the bottom of the code we updated the TrackActionStatus structure for the track we are fading out to mark it as in use and currently transitioning out. This way the system will not try to use this track for anything else (other than a track recovery) until it has been disabled and once again becomes a free resource.

We have now covered the blending case for the current set definition. We recovered a track if possible or recorded that this set definition will need to be assigned to a new track. We have also found any tracks from a previous action with the same group name and have set them to transition out.

The next section of code is similar and basically does the whole thing all over again, but this time for the non-blended case. If the current set definition is not set to blend then any track currently assigned to the same group will simply be halted and the new animation set will take over. Once again, if we do find a track that contains the exact same animation set and group name, we will recover that track and use it. This code is simpler than the blended case as we do not have to transition anything in or out.

Since it is very similar to the previous code we will discuss it only briefly. First we test to see if we have found a track that can be recovered. If so, we un-key any events (it may have been transitioning out) and update its corresponding TrackActionStatus structure to identify that the track is in use and not transitioning out. As before, our structure will store the address of the CActionDefinition object that applied this definition along with the definition index.

```
        // Non – Blending Case
        else
        {
            // Did we find an exact match for this group / set combination?
            if ( nSetMatched >= 0 )
            {
                // This is the same set as we're trying to apply, first thing is
                // kill all events currently queued for this track
                pActor->UnkeyAllTrackEvents( nSetMatched );

                // Clear status
                pTrackStatus = pStatus->GetTrackStatus( nSetMatched );
                pTrackStatus->bInUse           = true;
                pTrackStatus->bTransitioningOut = false;
                pTrackStatus->pActionDefinition = this;
                pTrackStatus->nSetDefinition    = i;

                // Get the current track description.
                pActor->GetTrackDesc( nSetMatched, &TrackDesc );


                // The user specified a set which is already active, so we will
```

```
                // retain the timing from this set, rather than the one we chose
                // above, if they specified only 'MatchGroup' mode.
                if ( pDefinition->TimeMode != ActTimeMode::MatchGroup )
                     TrackDesc.Position = pDefinition->fNewTime;

                // Setup other track properties
                TrackDesc.Speed  = pDefinition->fSpeed;
                TrackDesc.Weight = pDefinition->fWeight;

                // Set back to track
                pActor->SetTrackDesc( nSetMatched, &TrackDesc );

            } // End if found exact match

            else

            {
                // This definition is now due to be assigned to a new track
                pDefinition->bNewTrack = true;

            } // End if no exact match
```

The above code shows that if we can recover the track, we set the position, weight, and speed of the track immediately (no fading in or out) to that specified by the definition we are applying. Otherwise, if no match was found, we set the definition's bNewTrack Boolean to true so that we know at the bottom of the function we will need to find a free track to assign this set definition to.

In the next section of code we are still inside the non–blending code block. It essentially turns off (no fade out) any track which belongs to the same group as the definition we are currently applying. We wish to turn off any set which currently exists in the same group and apply our new one instead.

```
            // Find all tracks in this group, and kill them dead
            nCount = pActor->GetMaxNumTracks();
            for ( j = 0; j < nCount; ++j )
            {
                // Skip this track if it's our matched track
                // if we didn't find a match, nSetMatched will still be -1)
                if ( (signed)j == nSetMatched ) continue;

                // Retrieve action status information for this track.
                pTrackStatus = pStatus->GetTrackStatus( j );

                // Skip if this track is not in use
                if ( pTrackStatus->bInUse == false ) continue;

                // Does this belong to this group?
                if ( _tcsicmp( pTrackStatus->strGroupName,
                               pDefinition->strGroupName ) == 0 )
                {
                    // Kill any track events already setup
                    pActor->UnkeyAllTrackEvents( j );
                    pActor->SetTrackEnable( j, FALSE );

                    // Update status
```

```
                    pTrackStatus->bInUse            = false;
                    pTrackStatus->bTransitioningOut = false;

                } // End if matching group

            } // Next Track

        } // End if does not require blending

    } // Next Definition
```

After the definition loop has completed execution, the following will have happened for each definition.

1. We will have searched for any tracks currently assigned to the controller which belong to the same group

2. If one was found for a given definition, we recover the animation set, and set it to blend back in (or just turn it back on immediately in the non-blending case); otherwise we flag that we need to configure a new track.

3. Blend all other tracks out (or turn them off in the non blending case) that belong to the same group.

At this point, all we need to do is loop through the SetDefinitions array of this action one more time and test which ones have their bNewTrack Boolean members set to true. These are the ones that were not recovered and need new tracks.

```
    // Loop through each of our set definitions and assign to tracks if required
    for ( i = 0; i < m_nSetDefinitionCount; ++i )
    {
        // Retrieve the definition pointer for easy access
        pDefinition = &m_pSetDefinitions[i];

        // Skip if not required to assign to new track
        if ( !pDefinition->bNewTrack ) continue;
```

If we get here then we have found a definition that requires a new track so we will loop through all the tracks in our CActionStatus object and search for the first one that is not in use. Once we find one, we record the index in the local TrackIndex variable and break from the loop.

```
        // Find a free track
        long TrackIndex = -1;
        nCount = pActor->GetMaxNumTracks();
        for ( j = 0; j < nCount; ++j )
        {
            // Retrieve action status information for this track.
            pTrackStatus = pStatus->GetTrackStatus( j );

            // Free track?
            if ( pTrackStatus->bInUse == false ){TrackIndex=(signed)j; break; }

        } // Next Track
```

Now that we know the track on the mixer we wish to assign this definition too we will set up a D3DXTRACK_DESC structure to correctly set the attributes of the track to the properties of the definition (speed, weight, position, etc). Remember, we have already stored the time we wish the track position to be set to earlier in the function. In the next section of code, notice that we set the weight member of the track descriptor to the weight specified by the definition; we will set this to zero initially in a moment if this definition is configured to transition in.

```
        // Build track description
        TrackDesc.Position = pDefinition->fNewTime;
        TrackDesc.Weight   = pDefinition->fWeight;
        TrackDesc.Speed    = pDefinition->fSpeed;
        TrackDesc.Priority = D3DXPRIORITY_LOW;
        TrackDesc.Enable   = TRUE;
```

The track descriptor is set up, so let us also update the corresponding track in our CActionStatus object so that we know it is in use and know the name of the animation set and the group name currently assigned to that track.

```
        // Update track action status
        pTrackStatus = pStatus->GetTrackStatus( TrackIndex );
        pTrackStatus->bInUse           = true;
        pTrackStatus->bTransitioningOut = false;
        pTrackStatus->pActionDefinition = this;
        pTrackStatus->nSetDefinition    = i;

        _tcscpy( pTrackStatus->strSetName, pDefinition->strSetName );
        _tcscpy( pTrackStatus->strGroupName, pDefinition->strGroupName );
```

Now let us assign the physical animation set used by this set definition to the mixer track that we have chosen. Also, if the definition is set to blend, we will set the weight member of the track descriptor to zero initially and will set a sequencer event that will transition the weight to its desired strength over the mix length period. Remember, if a track currently shares the same group name as this definition, this track was already instructed to fade out over the same period earlier in the function.

```
        // Apply animation set
        pActor->SetTrackAnimationSetByName( TrackIndex, pDefinition->strSetName );

        // If we are blending, key events and override starting weight
        if ( pDefinition->BlendMode == ActBlendMode::MixFadeGroup )
        {
            TrackDesc.Weight = 0.0f;
            pActor->KeyTrackWeight( TrackIndex,
                                    pDefinition->fWeight,
                                    pActor->GetTime(),
                                    pDefinition->fMixLength,
                                    D3DXTRANSITION_EASEINEASEOUT );

        } // End if blending

        // Set track description
        pActor->SetTrackDesc( TrackIndex, &TrackDesc );
```

```
    } // Next Definition

    // Success!!
    return S_OK;
}
```

Notice at the bottom of the definition loop that we finally set the properties of the mixer track that we have just set up using the CActor::SetTrackDesc function.

That was a pretty intimidating function on first glance but fortunately there is nothing too complex going on. Rather, there are just lots of small tasks being performed which cumulatively makes everything appear complex.

We have now covered all the code to the new action system. We now have the ability to compose complex action sequences from combinations of animation sets in simple text files and have our application apply those actions with a single function call.

Before we leave the subject entirely, we need to discuss the small changes that have had to be made to the AttachController and DetachController methods in CActor. You will recall that these methods were introduced to allow our actor to be referenced. We must now make sure that the action system will also work correctly during referencing.

Safely Referencing Action Animating Actors

In Chapter Ten we wrote code that allowed multiple objects to reference the same actor, where each had different animation data or was at a different position on the animation timeline. This allowed us to instance the actor many times in the scene without having to load redundant copies of the actor geometry into memory. Our design solution was actually very simple. We decided that if multiple CObject's in our scene referenced the same actor, the actor would no longer own its own animation controller; instead, each object would manage its own controller. To update an object's animation, we simply attached its controller to the actor and then used the actor's methods to update it. Similarly, when rendering an object, we attached the object's controller to the actor and rendered the actor as normal. This solution works well and we will continue to use it.

The action system is also controller specific because the CActionStatus object maintains the current position of all the tracks on the controller. If we were to detach the controller and attach a different one, the CActionStatus object would be completely out of sync. Of course, we could write some function that would synchronize the CActionStatus object to a new controller whenever one is attached. While that would be easy to do, it would not be the wisest design choice. Remember, if we have multiple objects using the same actor, we want the actor's CActionStatus object to remember the settings of each reference because the controller itself will have no way of letting the action system know whether a track on the newly attached controller was in the process of fading out when it was last detached. We have essentially lost this information and our action system will not reference properly.

Of course, the solution is really rather straightforward. The CActionStatus object contains the state of a controller, and as each CObject referencing an actor owns its own controller, it should also own its own CActionStatus object in exactly the same way. All we need to do is add an extra parameter to our AttachController and DetachController methods that will allow us to pass/retrieve a CActionStatus object to/from a CActor object. That is, every time we attach an object's controller to an actor, its CActionStatus object will also be passed and will become the actor's current CActionStatus object. This way, each object owns both its own controller and the current state of that controller with respect to the action system. This allows the current state of an object's controller to persist during the attach/detach process.

CActor::Detach Controller

This method has been updated to take an optional CActionStatus parameter. If your object does not wish to use the action system we can just call this function as we always have. However, if you do wish to retrieve the current CActionStatus object of the actor, we can pass in the address of a CActionStatus pointer which, on function return, will pointer to the CActionStatus object the actor was using before you detached it.

Our application uses this function inside the CScene::ProcessReference function during loading. If the reference is to an X file that has already been loaded into an actor, we essentially switch the actor into reference mode. We fetch the controller and the CActionStatus from the actor and store it in the object instead. Then, any future references to this same actor causes the controller and the status object to be cloned and stored in a new object. We discussed the CScene::ProcessReference function in the animation workbook when we introduced this reference system. If you consult this function in Lab Project 12.2 you will see that virtually nothing has changed. All we do now is store the CActionStatus object of the actor in the CObject in addition to the controller.

Let us have a look at the detach controller function, which has been slightly updated. The function first makes a copy of the controller pointer and then sets the actor's controller pointer to NULL. It is now going to surrender its controller reference to the caller. At this point the actor has no controller anymore. We also copy the address of the actor's CActionStatus object in the passed pointer and set the actor's CActionStatus pointer to NULL as well. The actor no longer owns a CActionStatus object either. The function then returns the pointer to the new controller. The CActionStatus object is returned via the output parameter.

```
LPD3DXANIMATIONCONTROLLER CActor::DetachController(CActionStatus** ppActionStatus)
{
    LPD3DXANIMATIONCONTROLLER pController = m_pAnimController;

    // Detach from us
    m_pAnimController = NULL;

    // If the user requested to detach action status information, copy it over
    if ( ppActionStatus )
    {
        *ppActionStatus = m_pActionStatus;
        m_pActionStatus = NULL;
```

```
    } // End if requested action status

    // Note, we would AddRef here before we returned, but also
    // release our reference, so this is a no-op. Just return the pointer.
    return pController;

}
```

CActor::AttachController

This function also has a new parameter and a few new lines added at the bottom. The function now takes an additional third parameter allowing an action status object to be attached to the actor.

The first thing the function does is release the current action status object and controller being used by the actor. We then set the actor's controller and status object pointer to NULL.

```
void CActor::AttachController( LPD3DXANIMATIONCONTROLLER pController,
                               bool bSyncOutputs /* = true */,
                               CActionStatus * pActionStatus /* = NULL */ )
{
    // Release our current controller.
    if ( m_pAnimController ) m_pAnimController->Release();
    m_pAnimController = NULL;

    // Release our current action status (if provided).
    if ( m_pActionStatus ) m_pActionStatus->Release();
    m_pActionStatus = NULL;
```

If a new controller was passed, we will assign the address of this new controller to the actor's controller pointer and increase the reference count. If the bSyncOutputs parameter was set to true it means the caller would like the relative matrices of the hierarchy immediately updated to reflect the current position in the animation. We do this by calling AdvanceTime on the controller with a time delta of 0.0 seconds (we do not physically alter the position of the animation). The sequence of events is shown below.

```
    // Attach new controller if one is provided
    if ( pController )
    {
        // Store the new controller
        m_pAnimController = pController;

        // Add ref, we're storing a pointer to it
        m_pAnimController->AddRef();

        // Synchronize our frame matrices if requested
        if ( bSyncOutputs ) m_pAnimController->AdvanceTime( 0.0f, NULL );

    } // End if controller specified
```

In the final section of code we attach the passed CActionStatus object to the actor's member pointer and increase its reference count.

```
    // Attach new status object is one is provided
    if ( pActionStatus )
    {
        // Store the new status information
        m_pActionStatus = pActionStatus;

        // Add ref, we're storing a pointer to it
        m_pActionStatus->AddRef();

    } // End if status specified
}
```

Using the Action Animation System

In Lab Project 12.3 we use the action animation system to animate our game character. In this section we will detail the points of communication between the application and the actor in all matters involving the use of this action system.

Step 1: Updating CObject for Action Referencing

The first thing that has to be done is the expansion of our CObject structure to contain a CActionStatus pointer. As described in the previous section, this allows for proper actor referencing. Below we see the new CObject class as declared in 'CObject.h'.

```
class CObject
{
public:
    //-------------------------------------------------------------------------
    // Constructors & Destructors for This Class.
    //-------------------------------------------------------------------------
    CObject( CTriMesh * pMesh  );
    CObject( CActor   * pActor );
    CObject( );
    virtual ~CObject( );

    //-------------------------------------------------------------------------
    // Public Variables for This Class
    //-------------------------------------------------------------------------
    D3DXMATRIX                 m_mtxWorld;          // Objects world matrix
    CTriMesh                  *m_pMesh;             // Mesh we are instancing
    CActor                    *m_pActor;            // Actor we are instancing
    LPD3DXANIMATIONCONTROLLER  m_pAnimController;   // Controller
    CActionStatus             *m_pActionStatus;     // Action Status
};
```

Step 2: Loading the Player Model

In our application, the player controlled character uses the action system. Unlike previous demos, our CPlayer object will have a proper animating actor attached to it instead of the placeholder cube mesh we have been using in previous demos. This means the CGameApp::BuildObjects function which is called at application startup, will now need to call another function to load the model that will be attached to our CPlayer object.

The function we use to load and create the actor for the CPlayer object is part of the CScene namespace and is called CScene::LoadPlayer. This function accepts the name of an X file to load and the name of an .ACT file to load. The third parameter is the address of the CPlayer object that will be assigned the actor as its third person object.

We see this function being called from the CGameApp::BuildObjects function after the main scene has been loaded. It is highlighted in bold at the bottom of the following listing. The rest of this function should be familiar to you.

```
bool CGameApp::BuildObjects()
{
    CD3DSettings::Settings * pSettings = m_D3DSettings.GetSettings();
    bool                     HardwareTnL = true;
    D3DCAPS9                 Caps;

    // Should we use hardware TnL ?
    if ( pSettings->VertexProcessingType == SOFTWARE_VP ) HardwareTnL = false;

    // Release previously built objects
    ReleaseObjects();

    // Retrieve device capabilities
    m_pD3D->GetDeviceCaps(    pSettings->AdapterOrdinal,
                              pSettings->DeviceType, &Caps );

    // Set up scenes rendering / initialization device
    m_Scene.SetD3DDevice( m_pD3DDevice, HardwareTnL );

    ULONG LightLimit = Caps.MaxActiveLights;

    // Load our scene data
    m_Scene.SetTextureFormat( m_TextureFormats );

    // Attempt to load from IWF.
    if (!m_Scene.LoadSceneFromIWF( m_strLastFile, LightLimit ))
    {
        // Otherwise attempt to load from X file
        if (!m_Scene.LoadSceneFromX( m_strLastFile )) return false;

    } // End if failed to load as IWF

    // Attempt to load the player object (ignore any error at this point)
    m_Scene.LoadPlayer( _T("Data\\US Ranger.x"),
```

```
                        _T("Data\\US Ranger.act"),
                        &m_Player );

    // Success!
    return true;
}
```

To understand how the player object is loaded and configured we must take a look inside the new CScene::LoadPlayer method. The code to this new function is shown below with descriptions.

CScene::LoadPlayer

```
bool CScene::LoadPlayer(      LPCTSTR ActorFile,
                              LPCTSTR ActionFile,
                              LPCTSTR CallbackFile,
                              CPlayer * pPlayer )
{
    long nIndex = -1;

    // Add a new slot to the actor list
    nIndex = AddActor();

    // Allocate new actor
    CActor * pNewActor = new CActor;
    if ( !pNewActor ) return false;

    // Store the actor in the list
    m_pActor[ nIndex ] = pNewActor;
```

This function accepts the name of an X file to load for the player object, the name of an .ACT file to load describing the actions that can be performed by this actor, and a pointer to the CPlayer object. For the time being, we can ignore the third parameter (CallbackFile), since it will be discussed later.

We call CScene:AddActor to make space at the end of the scene's actor pointer array for a new actor. We like to store this in the scene's actor array with the rest of the scene actors so if anything goes wrong, the scene destructor will release it along with the rest of the actors it is managing. The AddActor method returns the array index where we should store our new actor pointer, so we allocate a new CActor object and store its pointer in the array at this position.

The next three lines are familiar. Before loading the actor from the X file we register the CALLBACK_ATTRIBUTE callback function, placing the actor into non-managed mode.

We also register a callback to handle the creation of callback keys during the actor's initial loading process. We will register the same function for this purpose as we did in Chapter Ten (CScene::CollectCallbacks). You will recall that when the actor has been loaded, this function will be called for each animation that was created. It gives the application an opportunity to fill an array of callback keys and return them to the actor. The actor then clones the animation sets with these callback keys registered. In Chapter Ten we used the callback system to register sound effects with the animation

sets. This function will be used for the same purpose again in this lab project, only a slightly more robust sound playing method will be used that requires some additional tweaking to the callback system.

```
// Load the actor from file
pNewActor->RegisterCallback(CActor::CALLBACK_CALLBACKKEYS,CollectCallbacks,this );
pNewActor->RegisterCallback(CActor::CALLBACK_ATTRIBUTEID,CollectAttributeID,this);

if ( FAILED(pNewActor->LoadActorFromX(  ActorFile,
                                        D3DXMESH_MANAGED,
                                        m_pD3DDevice,
                                        true,
                                        (void*)CallbackFile )) ) return false;
```

Notice that we now pass the CallbackFile pointer as a new parameter to the LoadActorFromX function. This will contain the name of another .ini file that contains the sound files we wish to play and the animation sets we wish them to apply to. The actor will pass this to its ApplyCallbacks function after it has been loaded. The ApplyCallbacks function will then call the CScene::CollectCallbacks function (the registered callback) for each animation set loaded from the X file, passing in this string. The callback key creation function can then open the .ini file and read in the sound effects for the current animation set be created. These sounds are then registered as callback keys with the animation set. This will all be discussed at the end of this section. Just know for now that the pointer in the LoadActorFromX parameter list is pumped straight through to any registered CALLBACK_CALLBACKKEYS function to aid in the creation of the callback keys.

Now that the actor has been fully loaded along with any animation data that may exist, we can load the action data from our ACT file into our new actor. We will also extend the limits of the actor's controller so that the action system has 20 mixer tracks to play with when performing its fades between actions. The CActor::SetActorLimits method is used for this purpose. It ultimately issues a call to the CActor::CActionStatus::SetMaxTrackCount method so the status object's TrackActionStatus array is resized to retain a 1:1 mapping with mixer tracks available on the controller.

```
    // Load the action definition file
    if ( FAILED(pNewActor->LoadActionDefinitions( ActionFile )) ) return false;

    // Set the actor limits to something reasonable
    if ( FAILED(pNewActor->SetActorLimits( 0, 20 )) ) return false;
```

At this point, our actor has been correctly configured and contains all its geometry, animation, and action data. We now need to assign this actor to an object. To do this we use the CScene's AddObject method to make space at the end of its CObject array for a new object pointer. Then we allocate a new CObject, store its pointer in the object array, and assign our new actor to its m_pActor member. Finally, before returning, we call the CPlayer::Set3rdPersonObject method to store a pointer to this CObject in the CPlayer. This is the object that the CPlayer::Render method will use to render itself.

```
    // Add a new slot to the object list
    nIndex = AddObject();

    // Create a new object
    CObject * pNewObject = new CObject;
```

```
    if ( !pNewObject ) return false;

    // Store the object in the list
    m_pObject[ nIndex ] = pNewObject;

    // Store the object details
    pNewObject->m_pActor = pNewActor;

    // Inform the player of this object
    pPlayer->Set3rdPersonObject( pNewObject );

    // Success!
    return true;
}
```

We have now seen the adjustments that we have had to make to our application to support the loading of the action data and the population of an actor with this action data. In the next section we will discuss how the CPlayer object has been modified to apply the relevant actions to its attached actor.

Step 3: Applying Actions

You will hopefully recall that when we first introduced our CPlayer object in Chapter Four of Module I, the CPlayer::Update function was called to update the position of the player in response to user input.

In every iteration of our game loop, the CGameApp::ProcessInput function is called to determine if the player is pressing any keys or moving the mouse. In response to the movement keys being pressed for example, this function would call the CPlayer::Move function to update the velocity variable in the player object. If the mouse had been moved with the left button depressed, the CPlayer::Rotate method would be called to apply a rotation to the CPlayer's world matrix. However, the position of the player was not updated until the very end of that function when the CPlayer::Update method was called. It was this function that was responsible for updating the position of the player and its attached camera based on the currently set velocity of the player. Other factors came into play as well such as resistance forces working against the velocity vector.

Although we will not show the code to the CPlayer::Update function again here, one new function call has been added to it that will call a function called CPlayer::ActionUpdate. In this function, the player object has a chance to apply any actions it wishes based on user input or game events. This method will also be responsible for building the world matrix of the attached CObject that houses the actor.

Before we look at the code to the ActionUpdate method, we will discuss what the code is assumed to be doing in this example. Also note that the actual function code contained in Lab Project 12.3 may be different from what is shown next. At the time of writing it was still possible that different assets may be used in our final demo. Since the ActionUpdate function is largely dependant on the animation assets being used (e.g., the names of animation sets, etc.) the function code shown here is to be used only as an example of what an ActionUpdate function can look like.

In this example we will assume that the .ACT file contains the following five actions. Each action is comprised of three sets for the torso, hips, and legs respectively. We can forget about the 'Hips' set in each action as it essentially just undoes the pelvis rotation caused by walking.

- **Walk At Ease**
  In this action, the torso animation set adopts a typical walking sequence. The arms are swinging back and forth in tandem with the legs. The weapon is loosely held in one of the hands and is also swinging back and forth. The legs animation set is performing a typical footsteps animation. In this action, the character is certainly very relaxed and is not expecting any hostile activity.

- **Walk Ready**
  In this action the character is walking on patrol and is ready for anything. The torso animation set now has the gun held firmly to the character's chest so that he can assume the shooting pose at a moment's notice. The legs animation set is identical in all three walk actions – a standard footsteps animation.

- **Walk Shooting**
  When this action is applied the same legs animation is used as in the other two walk cases but the torso is now playing an animation set that holds the gun up to the soldier's shoulder in a targeting pose looking for enemies to fire at.

- **Idle Ready**
  This action will be played when the character is not currently walking and the fire button is not being pressed. The legs animation set simply has the legs in a stationary position so that the character is standing still. Whenever the character is idle, he will assume the ready pose with his torso (i.e., the animation set that has the character holding the gun to his chest).

- **Idle Shooting**
  This action uses the same torso animation set as the Walk Shooting action, but uses the same legs animation set as the Idle Ready action. This action is applied when the player presses the fire button (right mouse) but the character is not currently being moved at the same time. The character stands still, looking through the sight of his gun for hostile targets.

We can imagine when most of these actions will need to be applied. For example, if the player is currently moving and the fire button is pressed, then the 'Walk Shooting' action will be played. If the player stops moving but is still shooting, then the 'Idle Shooting' action will be applied. If the character is walking and his weapon has not been fired, we will apply the 'Walk At Ease' action. So when does the 'Walk Ready' action get applied?

In order to make the character feel a little more lifelike, the 'Walk Ready' action is played for five seconds after the fire button has been pressed and released. If we imagine that the character is currently moving, and we are playing the 'Walk At Ease' action, assume that the fire button is pressed and the 'Walk Shooting' animation is applied (which remains applied until the fire button is released). Instead of simply returning to the 'Walk At Ease' action as soon as the fire button is released, we apply the 'Walk Ready' action for 5 seconds. Assuming the fire button is not pressed again, after five seconds the character will drop his arms into the 'at ease' posture and start walking normally. This makes the

character seem a little more intelligent. In real life, a soldier would hardly assume an ‘At Ease’ posture immediately after firing a shot. He would probably fire and then remain in the ready position for some time, ready to fire again until he is sure that the threat has passed. So we add a little bit of that logic here just to demonstrate how you can use the AAS to accomplish different design ideas.

With these animation assets now explained, we are ready to examine the code to this function that implements the behavior described above.

The function is passed one parameter by the CPlayer::Update method describing the elapsed time in seconds between the last update and the current one. Here is the first section of the function that builds the world matrix for the CPlayer’s attached object. The player currently stores a right vector, up vector, look vector and a position vector, so building the matrix is a simple case of storing these vectors in the rows of the attached CObject’s world matrix.

```
void CPlayer::ActionUpdate( float TimeScale )
{
    ULONG i;

    // If we do not have a 3rd person object, there is nothing to do.
    if ( !m_p3rdPersonObject ) return;

    // Get the actor itself
    CActor * pActor = m_p3rdPersonObject->m_pActor;

    // Update our object's world matrix
    D3DXMATRIX * pMatrix = &m_p3rdPersonObject->m_mtxWorld;

    pMatrix->_11=m_vecRight.x; pMatrix->_21=m_vecUp.x; pMatrix->_31=m_vecLook.x;
    pMatrix->_12=m_vecRight.y; pMatrix->_22=m_vecUp.y; pMatrix->_32=m_vecLook.y;
    pMatrix->_13=m_vecRight.z; pMatrix->_23=m_vecUp.z; pMatrix->_33=m_vecLook.z;
    pMatrix->_41 = m_vecPos.x;
    pMatrix->_42 = m_vecPos.y;
    pMatrix->_43 = m_vecPos.z;
```

As you can see in the above code, if the CPlayer has no third person object attached to it we simply return. Otherwise we fetch a pointer to the object’s CActor and the object’s world matrix. We then build the world matrix for the third person object.

In the next section of code we will define some static variables.

```
    static float fReadyTime = 200.0f;
    static bool  bIncreaseReadyTime = false;
```

The fReadyTime is set to an arbitrarily high value of 200 seconds and will be used as a timer to determine how long the ‘ready’ action should remain applied before the character is put back into an ‘at ease’ action state. Whenever the fire button is pressed, fReadyTime will be set to 0 (zero seconds) and the bIncreaseReadyTime boolean will be set to false so that we will not increase this timer while the fire button is being pressed. As soon as the fire button is no longer being pressed, the boolean will be set to true and the timer will be incremented each time the function is called. As this timer is always set to zero seconds when the player fires, as soon as the player is no longer firing, we will apply the ‘ready’ pose

for five seconds (i.e., while fReadyTimer is less than 5.0). As soon as its value is incremented past five seconds we know that we have been in the ready pose long enough and should now apply the ‘at ease’ pose. This cycle repeats whenever the fire button is pressed and the timer is set back to zero.

We get the speed of the character by calculating the length of the CPlayer’s velocity vector. You will see why this is needed in a moment. We then use the GetKeyState Win32 function to test if the left mouse button is being pressed. If so, we wish to apply a shooting action and reset the ready timer to zero.

```
    // Calculate movement speed
    float fSpeed = D3DXVec3Length( &m_vecVelocity );

    // In Shooting State?
    bool bShooting = false;
    if ( GetKeyState( VK_LBUTTON ) & 0xF0 )
    {
        // We are shooting
        bShooting          = true;

        // Setup timing states
        fReadyTime         = 0.0f;
        bIncreaseReadyTime = false;

    } // End if left button is pressed
    else
    {
        // Set our 'ready' timer going
        bIncreaseReadyTime = true;

    } // End if left button is not pressed
```

At the end of this loop, if we are pressing the fire button the timer will be reset to zero and the bIncreaseReadyTime boolean will be set to false indicating that we do not wish to increment the ready timer in this update and should leave it at zero. Otherwise, if the fire button is not being pressed the Boolean will be set to true, meaning a timer increment should happen. We also set the bShooting boolean to true if the fire button is being pressed so that we will know later in the function that an action should be applied that includes a torso shooting pose.

We now test the current value of the timer and if it is found to contain less than 5 seconds of elapsed time since being reset, we should be in the ready pose. So we set the bReadyState boolean to true. Otherwise, it is set to false. We also increment the ‘ready’ timer using the elapsed time passed into the function. This timer increment only happens if the fire button is not being pressed.

```
    // In ready state?
    bool bReadyState = false;
    if ( fReadyTime < 5.0f ) bReadyState = true;
    if ( bIncreaseReadyTime ) fReadyTime += TimeScale;
```

At this point we have determined what the torso of the character should look like. He is either shooting (bShooting=true), in the ready pose (bReadyState=true), or is walking at ease. Next we have to find out

what the legs of the character should be doing. There are two choices, they are either walking or standing still.

We decided to place the character into an idle legs pose if the speed of the character drops to less than 6 units per second. This is a value that you will want to tweak depending on the scale of your level, the resistance forces you apply, etc. In our case, 6 units of movement per second represents a very small amount of movement at the per frame level and will be cancelled out by friction fairly quickly if no additional force is applied. Therefore, as the player slowly comes to a halt, we place its legs into the idle pose slightly prematurely. This looks more natural than the character stopping while his legs are still spread apart in a walk pose, and then having the legs slide into the idle pose on the spot.

```
    // Idling?
    bool bIdle = false;
    if ( fSpeed < 6.0f ) bIdle = true;
```

We now know whether the legs are idle or not, so we can determine which action to play. If our legs are idle and we are shooting, we should apply the ‘Idle1_Shooting’ action. If the fire button is not being pressed but the legs are still idle we should play the ‘Idle1_Ready’ action.

```
    // Apply the correct action
    if ( bIdle )
    {
        // What is our pose?
        if ( bShooting )
            pActor->ApplyAction( _T("Idle1_Shooting") );
        else
            pActor->ApplyAction( _T("Idle1_Ready") );

    } // End if idling
```

That takes care of the idle case. Next we have the code block that deals with the non-idle case which surrounds the remaining code in this function. If our legs are not idle and we are shooting, we will play the ‘Walk_Shooting’ action. If the fire button is not pressed but the ‘ready’ timer contains a value of less than 5 seconds, we will apply the ‘Walk_Ready’ action. Otherwise, we will play the ‘Walk_At_Ease’ action.

```
    else
    {
        // What is our pose?
        if ( bShooting )
            pActor->ApplyAction( _T("Walk_Shooting") );
        else
            if ( bReadyState )
            pActor->ApplyAction( _T("Walk_Ready") );
        else
            pActor->ApplyAction( _T("Walk_At_Ease") );
```

At this point we have applied the correct walking action to the actor, but we should really adjust the speed of the tracks used by this action so that it matches the speed of the character. It would look very

strange if the legs and arms of the character animated at a constant speed without accounting for the actual speed of the character's forward motion.

We will calculate the animation speed by the taking the dot product between the velocity vector and the look vector of the player since this describes to us how much of the velocity is attributed to the fact that the player is walking forward. If the velocity and look vectors are aligned, a value of 1 will be returned. This will become smaller as the vectors become misaligned. Once we have the value of the dot product (which is the cosine of the angle scaled by the length of the velocity vector), we scale it by 0.027. Finding this scaling value was just a trial and error process and you may wish to change it to suit your scene scale.

```
     // Calculate the speed of the tracks
     float fDot   = D3DXVec3Dot( &m_vecLook, &m_vecVelocity );
     float fSpeed = fDot * 0.027f;
```

Finally, we will loop through each track in the actor's CActionStatus object and examine the TrackActionStatus structure. This will tell us if the track is currently in use and the action that assigned that animation set to the mixer track. We are searching for all tracks that are currently in use and that were set up by the current CActionDefinition object that we just applied. If one is found, we set the speed of that track on the mixer to the speed value just calculated.

```
     CActionStatus * pStatus = pActor->GetActionStatus();
     CActionDefinition * pDefinition = pStatus->GetCurrentAction();

      for ( i = 0; i < pStatus->GetMaxTrackCount(); ++i )
      {
       CActionStatus::TrackActionStatus *pTrackStatus=pStatus->GetTrackStatus(i);

       // Skip if this track is not in use
       if ( !pTrackStatus->bInUse ) continue;

       // Skip if this track wasn't assigned by the currently active definition
       if ( pTrackStatus->pActionDefinition != pDefinition ) continue;

       // Set the track speed
       pActor->SetTrackSpeed( i, fSpeed );

      } // Next Track

   } // End if walking
}
```

Since we are looping through all the tracks, we are guaranteed to find those that were set up by the currently applied action. This means we will update the speed of the torso, hips, and legs tracks for the current action simultaneously ensuring that the sets within a given action remain synchronized.

We have now covered the action system in its entirety. We have not only covered the code to the action system itself, but we have also examined how the application uses it to animate a game character. As mentioned, the actual contents of the CPlayer::ActionUpdate method may be quite different from the one shown above. Indeed, this is a function that will probably have its contents changed quite frequently

to suit the assets and animation sets that you use in your own game projects. When we cover scripting in Module III, much of this code will go away and we can configure animation changes offline without having to update and recompile source code. But that is for another day.

The remainder of this workbook will be spent detailing other upgrades we have made to our application framework which are not specifically related to the Action Animation System.

Adding Multiple Sound Effect Call-Back Keys

In Chapter Ten we implemented a lab project that had sound effects that were triggered by the actor's animation sets via the callback system. The problem is, when the callback handler played those sounds in response to a callback event being triggered, it used the Win32 'PlaySound' function. To be sure, this is not a course on sound and we do not want to get caught up in the specifics of the DirectSound API (which we are going to use), but the Win32 PlaySound function is simply too limiting to continue using it moving forward. The biggest issue is that it only allows us to play one sound at a time. That means, if we wish to play the sound of our characters footsteps as he walks and we wish to have an ambient background sound, such as blowing wind, we are out of luck. This is far too restrictive, so we will also use this lab project to introduce a new way to play sounds, via two very helpful classes that ship with the DirectX SDK framework.

Using CSoundManager

If you look in the "/Sample/C++/Common/" folder of your DirectX 9 installation directory you will see two files named DXUTSound.h and DXUTSound.cpp. These files contain the implementation of a class called CSoundManager that we can include in our projects to easily play (multiple) sounds. This class and the class it uses to represent individual sounds (CSound) wrap the DirectSound API, making it delightfully easy to load and play sound. Although the CSoundManager class exposes many functions to perform techniques like 3D positional sound, at the moment we are only interested in using this class to load and play simple sound. As such, we will only use a very small subset of the functions available.

So that we do not get caught up in the code to these classes (which are way off topic in a graphics course) we have compiled them into a .lib file which is included with this project. These files are called libSound.lib and libSound.h and they are located in the Libs directory of the project. CGameApp.h and CScene.cpp both include this header file, giving them access to the CSoundManager class.

```
#include "..\\Libs\\libSound.h"
```

The CSoundManager object is used to create CSound objects that can be played. As such, we have made it a member variable of the CGameApp class.

```
CSoundManager     m_SoundManager;
CSoundManager     * GetSoundManager     ()       { return &m_SoundManager; }
```

Notice that we have also added a new inline member function called GetSoundManager that returns a pointer to the sound manager object. This allows other modules (such as CScene for example) to retrieve the application's sound manager object and use it to create new sounds during the registration of callback key data with the actor.

As we know, the CGameApp::InitInstance method is called when the application is first started. In turn it calls a number of functions to configure the state of the application prior to the commencement of the main game loop. This method calls functions such as CreateDisplay and TestDeviceCaps and the BuildObjects method which actually loads the geometry. Since the CSoundManager object requires some initialization of its own, we have added a new method to the CGameApp class called CreateAudio. Below we show how this function is called from the InitInstance function. The new line is highlighted in bold.

```
bool CGameApp::InitInstance( HANDLE hInstance, LPCTSTR lpCmdLine, int iCmdShow )
{
    // Store the initial starting scene
    m_strLastFile = _tcsdup( _T("Data\\Landscape.iwf") );

    // Create the primary display device
    if (!CreateDisplay()) { ShutDown(); return false; }

    // Create the primary audio device.
    if (!CreateAudio()) { ShutDown(); return false; }

    // Test the device capabilities.
    if (!TestDeviceCaps( )) { ShutDown(); return false; }

    // Build Objects
    if (!BuildObjects()) { ShutDown(); return false; }

    // Set up all required game states
    SetupGameState();

    // Setup our rendering environment
    SetupRenderStates();

    // Success!
    return true;
}
```

The CreateAudio method contains only three lines of code to call methods of the CSoundManager object to initialize it. This involves informing the operating system of the control we would like over the sound hardware and the format in which we would like sounds to be played. Here is the code to the function.

```
bool CGameApp::CreateAudio()
{
    // Create the audio devices.
    if( FAILED(m_SoundManager.Initialize( m_hWnd, DSSCL_PRIORITY ))) return false;
    if( FAILED(m_SoundManager.SetPrimaryBufferFormat( 2 ,22050,16))) return false;

    // Success!
    return true;
}
```

In the first line we call the CSoundManager::Initialize function, passing in the application's main window handle and the flag DSSCL_PRIORITY. When using a DirectSound device with the priority cooperative level (DSSCL_PRIORITY), the application has first rights to audio hardware resources (e.g., hardware mixing) and can set the format of the primary sound buffer. Game applications should use the priority cooperative level in almost all circumstances. This level gives the most robust behavior while allowing the application control over sampling rate and bit depth.

In the second line we have to inform the sound manager of the format of the sounds we would like it to create and play in its primary buffer. We can think of the primary buffer as being very much like the audio equivalent of a 3D device's frame buffer. It is where all of the individual sounds that are currently playing get blended together to produce the final audio output.

As the first parameter to the CSoundManager::SetPrimaryBufferFormat method, we pass in the number of channels we would like the primary buffer to have. We pass in a value of 2 because we would like it to play sounds in stereo. A stereo sample has two separate channels of audio within the same .wav file (it has separate audio streams for the left and right speakers). In short, if we have stereo sound samples that we wish to load and play, this will ensure that the DirectSound device will be able to reproduce the final stereo output. If you wanted to play only mono sounds, you could pass 1 for this parameter. A mono sound has only one channel of audio data which gets duplicated in each speaker.

As the second parameter we pass in a sample rate of 22050, which means we are setting the quality at which it plays back sounds to 22khz (pretty fair quality). The third parameter specifies that we would like the primary buffer to be able to play back sounds in 16-bit sound quality. 16-bit sound has much nicer audio quality then 8-bit sounds. Of course, if your sound samples have been recorded at 8-bit to begin with, then they are still going to sound bad on a 16-bit quality device. However, this setting means that 16-bit sound samples will not have to be played back at 8-bit resolution.

When the above function returns, the sound manager will be ready for action and the application can use it to load and create sounds. From this point on, the only method we will be interested in calling is the CSoundManager::Create method. This method is defined as shown below. Actually, the function has many optional parameters after the two shown here, but these are the only two we are going to use, so we will worry about them. This function is defined in DXUTSound.h.

HRESULT Create(CSound ppSound, LPWSTR strWaveFileName)**

The first parameter is the address of a CSound object pointer which will point to a valid CSound object on function return. The second parameter is a string containing the name of the wav file we would like to load. When the function returns, the CSound object will encapsulate that sound effect and we can use its CSound::Play method to play the sound. We can create as many CSound objects as we want just by calling this function over and over again for the different sound effects we wish to load.

Below, we see an example of creating a CSound object for a wav file called 'footsteps.wav'.

```
CSound * pSound = NULL;
m_pSoundManager->Create( &pSound, "footsteps.wav" );
```

At this point the sound would be loaded and would be ready to play. Whenever we wish to play that sound we can simply do the following:

```
pSound->Play( )
```

As you can see, it is delightfully easy to use. If we play a sound and a sound is already playing, they will be mixed in the primary buffer of the DirectSound device and both sounds will heard simultaneously. In fact, there is no hard limit on the number of sounds that can be played simultaneously, although you will see a drop in game performance if you try to play too many. When multiple sounds are being played, they all have to be mixed in the primary buffer, so there is some associated overhead.

We can also pass parameters to the CSound::Play method to tell it how we would like it to play. For example, we might specify that we would like the sound to play repeatedly (i.e., loop). The code snippet shown below comes from the very bottom of the CScene::LoadSceneFromIWF function in Lab Project 12.3. It creates and plays a looping wind sound effect while the application is running. Notice how we query the CGameApp class for its CSoundManager pointer and then use it to load a wave file called 'Wind Ambient.wav'. We then play the sound immediately.

Excerpt from CScene.cpp : LoadSceneFromIWF

```
CSoundManager * pSoundManager = GetGameApp()->GetSoundManager();
if ( pSoundManager )
{
  // Release the old sound if it's resident
  if ( m_pAmbientSound ) delete m_pAmbientSound;

  // Create the ambient sound
  pSoundManager->Create(&m_pAmbientSound,_T("Data\\Sounds\\Wind Ambient.wav"));
  m_pAmbientSound->Play( 0, DSBPLAY_LOOPING );

} // End if application has sound manager.
```

Notice that this time we are passing in two parameters to the Play function. The first is the priority of the sound. We pass in the lowest (default) priority of 0, stating that this sound is no more important than any other sound our application might play. The second parameter is a flag that describes how the DirectSound buffer (all sounds are loaded into their own buffers) should be played. There is only one modifier flag that can be used at the moment: DSBPLAY_LOOPING. This instructs DirectSound to play the sound in a loop. The absence of this flag means the sound will play once and then stop.

Now that we have the ability to play multiple sounds, we will need to write a new (CScene::CollectCallbacks) function that uses the sound manager to create and load these CSound objects and store them in callback keys in the actor's animation sets. Additionally, it would be nice if we could extract the sound data from another type of .ini file rather than hard coding the CollectCallbacks function to load the files we wish to play. In the next section we will discuss some upgrades to the actor's callback key collection system that will allow the callback key collection function (CScene::CollectCallbacks) access to the name of the file that contains the descriptions of the sound files we wish to load.

The CollectCallbacks Callback Function

Before we discuss any changes, let us just refresh our memories with respect to the actor's callback key collection system.

As we know, when the LoadActorFromX method is called, it issues a call to the D3DXLoadMeshHierarchyfromX function to build the actor's hierarchy and its animation controller (if applicable). As we discussed in Chapter Ten, we are unable to register callback keys with an animation set that has already been created. Therefore, we implemented a private method of the actor called ApplyCallbacks which (when called from LoadActorFromX) would loop through every animation set that was loaded and call an application registered callback function (CScene::CollectCallbacks). This function would be passed an array of callback keys which it could fill with information. The application would register a special type of callback function with the actor prior to calling its LoadActorFromX function. This callback function would then be called for each loaded animation set. The callback function could create callback keys and pass them back to the actor. The actor would then clone the original animation set into a new one that has the callback keys registered with it.

You will remember that originally, the CScene::CollectCallbacks function loaded some wave files and stored them in CActionData objects. The pointers to these objects were then stored in the callback key itself as context data. When the callback was triggered, this pointer would be passed to the callback handling function which could then access the wave file stored inside and play it. You will also recall that the CActionData class was derived from IUnknown so that the actor could safely release objects of this type when the actor's controller was being destroyed.

This all worked fine, but when we have multiple sounds to load it can certainly become cumbersome to have to hard code the loading of each sound in the callback key's collection function. It would much nicer if we could store the sound files and the animation sets they should be registered for in an external file. That is exactly what we do in this lab project. You will find a file in the Data directory of this lab project called 'US Ranger.cbd' which contains the sound data that should be registered with the Walk_Legs_Only animation set. We will have a look at the contents of this file a little later on. For now, just know that it contains the names of sound files that should be registered with a particular animation set as callback keys.

> **Note:** The .cbd extension that we have given to our file is short for 'CallBack Data'. Although we are using this file to store sound data, it can be used to store any type of callback key data we wish to represent.

This raises an interesting question. If our CScene::CollectCallbacks function is going to create the callback information from a file, how does it know the name of that file, which may be different for different actors? The answer is simple -- we have to give the actor's COLLECTCALLBACKS callback function prototype an additional parameter so that other data can be passed by the actor to the callback function that creates the callback keys. This pointer can be used for anything, but in our code we use it to pass the name of the .cbd file containing the sound effect properties.

If we look in CActor.cpp we are reminded that the actor stores the pointer for a registered callback function of type CALLBACK_CALLBACKKEYS in the fourth element of its callback function array.

```
enum CALLBACK_TYPE { CALLBACK_TEXTURE = 0,
                     CALLBACK_EFFECT = 1,
                     CALLBACK_ATTRIBUTEID = 2,
                     CALLBACK_CALLBACKKEYS = 3,
                     CALLBACK_COUNT = 4 };
```

That is, if we wish to register a function 'SomeFunction' as a callback key collection function, we would do so like this:

```
pNewActor->RegisterCallback(CActor::CALLBACK_CALLBACKKEYS,somefunction,this );
```

As discussed in the past, the first parameter describes the type of callback we wish to register with the actor. In this example, it is a callback to create callback keys during actor creation. This will be called by the actor just after the animation data has been loaded, once for each animation set so the function has a chance to register keys with it. The second parameter is the function name itself, which in this example is called 'somefunction'. The third parameter is optional context information that will be passed through to the callback key creation function when it is called. In this example the 'this' pointer is being passed, which means the callback function will have access to member variables of the object that is registering the function. Remember, callback functions must be static if they are methods of a class.

Of course, the function signature for 'somefunction' must be defined in a very specific way as dictated by the actor. Previously, this function had to return a ULONG (the number of callback keys it placed into the passed array) and accept the four parameters shown below.

```
typedef ULONG (*COLLECTCALLBACKS )( LPVOID pContext,
                                    LPCTSTR strActorFile,
                                    LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                    D3DXKEY_CALLBACK pKeys[] )
```

Every time this function is called by the actor, it will be passed the context that was passed into the RegisterCallback method (the *this* pointer for example), the name of the actor that is calling it, the name of the animation set that we are providing keys for, and a callback key array which we must fill with any callback keys we would like to register. This array is owned by the actor which calls this function just after the X file has been loaded.

When we look at the parameter list we can see that it provides no way for the name of the (.cbd) sound file to be passed. Therefore, we will modify the signature of this callback function (in CActor.h) to contain an additional parameter, as shown below.

```
typedef ULONG (*COLLECTCALLBACKS )( LPVOID pContext,
                                    LPCTSTR strActorFile,
                                    void * pCallbackData,
                                    LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                    D3DXKEY_CALLBACK pKeys[] )
```

As you can see, the actor now has the ability to pass an extra piece of information to the CollectCallbacks function in the form of a void pointer (pCallbackData). Our player's actor will use this to pass the name of a .cbd file to the callback function so that it knows where to load its sound data

from. The problem is, how does the actor know what data we want to send into this function? This too is now optionally passed into the LoadActorFromX method via a new parameter on the end.

```
HRESULT             LoadActorFromX  (     LPCTSTR FileName,
                                          ULONG Options,
                                          LPDIRECT3DDEVICE9 pD3DDevice,
                                          bool ApplyCustomSets = true,
                                          void * pCallbackData = NULL );
```

As you can see, when we call the actor's LoadActorFromX method we now have the option of passing in an additional void pointer. When the animation data has been loaded, the LoadActorFromX function will pass this pointer to the CActor::ApplyCallbacks method. This function originally took no parameters, but has been altered now so that it accepts a void pointer:

```
HRESULT ApplyCallbacks          ( void * pCallbackData );
```

As we know, this function loops through every loaded animation set in the actor, and if the CALLBACK_CALLBACKKEYS callback function has been registered, it will be called, passing in this new parameter as well as those shown above in the COLLECTCALLBACKS function prototype.

This means that the static CScene function that we register with the actor for callback key creation must also have this additional parameter added, so that it matches the layout defined for COLLECTCALLBACKS callback function. We will look at the actual code to the function later but below we see that its parameter list has been changed to match the correct signature for a callback key creation callback function.

```
ULONG CScene::CollectCallbacks(     LPVOID pContext,
                                    LPCTSTR strActorFile,
                                    void * pCallbackData,
                                    LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                    D3DXKEY_CALLBACK pKeys[] )
```

Representing our Sound Data

In Chapter Ten the CollectCallbacks function stored the names of the sound files we wanted to play in a class called CActionData. This was a very simple class that stored a string containing the name of the wave file. It was a pointer to an object of this type that was stored in the callback key so that it could be passed to the callback handler when the callback key was triggered. The callback key handler could then extract the sound file name and load and play it. This object also had to be derived from IUnknown and implement the AddRef, Release, and QueryInterface methods so that the actor could safely release these objects when the animation set was about to be destroyed. Although the actor knows nothing about the type of data we store in a callback key as context data, as long as it knows it is derived from IUnknown, before releasing the animation set it can fetch a pointer to each callback key's context data pointer, cast it to an IUnknown pointer and then call its Release method. This instructs the CActionData object (or

any IUnknown derived object) to destroy itself without the actor needing to have knowledge of what the object actually is and how its memory should be released.

The CActionData class served us well for previous demos, but now we will replace it with something a little more flexible. In CScene.h we declare an abstract base interface called IActionData to act as the base class from which any callback key context objects we register will be derived. This is a pure abstract class so it cannot be instantiated; it just sets out the methods that each callback key object must support.

The interface's GUID and definition is inside CScene.h, as shown below.

```
const GUID IID_IActionData    = {0xCCB7147E, 0x4480, 0x4C08, {0x8F, 0xC6, 0x95,
                                 0x57, 0xAC, 0x59, 0xA3, 0xA}};

class IActionData : public IUnknown
{
public:
      virtual HRESULT CreateCallback(     ULONG Index,
                                          LPVOID pContext,
                                          LPCTSTR strActorFile,
                                          void * pCallbackData,
                                          LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                          D3DXKEY_CALLBACK * pKey ) = 0;

      virtual HRESULT HandleCallback( UINT Track, CActor * pActor ) = 0;
};
```

As you can see, we now insist that not only should a callback object support the methods from IUnknown for lifetime encapsulation purposes, but it must now implement two methods called CreateCallback and HandleCallback. In order to understand the parameter lists to these functions, let us look at what the .cbd file looks like that we created for Lab Project 12.3.

Excerpt from US Ranger.cbd

```
[Walk_Legs_Only]

CallbackCount               = 4

; First Callback (Left foot touches ground)
Callback[0].Ticks           = 20
Callback[0].Action          = PLAY_SOUND
Callback[0].PlayMode        = RANDOM
Callback[0].SampleCount     = 4
Callback[0].Sample[0].File  = Data\Sounds\Footsteps L1.wav
Callback[0].Sample[1].File  = Data\Sounds\Footsteps L2.wav
Callback[0].Sample[2].File  = Data\Sounds\Footsteps L3.wav
Callback[0].Sample[3].File  = Data\Sounds\Footsteps L4.wav

; Second Callback (Right foot touches ground)
Callback[1].Ticks           = 40
Callback[1].Action          = PLAY_SOUND
Callback[1].PlayMode        = RANDOM
Callback[1].SampleCount     = 4
```

```
Callback[1].Sample[0].File  = Data\Sounds\Footsteps R1.wav
Callback[1].Sample[1].File  = Data\Sounds\Footsteps R2.wav
Callback[1].Sample[2].File  = Data\Sounds\Footsteps R3.wav
Callback[1].Sample[3].File  = Data\Sounds\Footsteps R4.wav

; First Callback (Left foot touches ground)
Callback[2].Ticks           = 60
Callback[2].Action          = PLAY_SOUND
Callback[2].PlayMode        = RANDOM
Callback[2].SampleCount     = 4
Callback[2].Sample[0].File  = Data\Sounds\Footsteps L1.wav
Callback[2].Sample[1].File  = Data\Sounds\Footsteps L2.wav
Callback[2].Sample[2].File  = Data\Sounds\Footsteps L3.wav
Callback[2].Sample[3].File  = Data\Sounds\Footsteps L4.wav

; Second Callback (Right foot touches ground)
Callback[3].Ticks           = 80
Callback[3].Action          = PLAY_SOUND
Callback[3].PlayMode        = RANDOM
Callback[3].SampleCount     = 4
Callback[3].Sample[0].File  = Data\Sounds\Footsteps R1.wav
Callback[3].Sample[1].File  = Data\Sounds\Footsteps R2.wav
Callback[3].Sample[2].File  = Data\Sounds\Footsteps R3.wav
Callback[3].Sample[3].File  = Data\Sounds\Footsteps R4.wav
```

The layout of this file is very simple, especially in this case where we are only defining the information for one animation set.

At the top of the file we can see that the only block defined is for the animation set called ‘Walk_Legs_Only’. This entire file is just defining a number of callbacks for the one set. If we had callback key data to define for multiple animation sets, there would be multiple sections in this file arranged one after another.

Inside the Walk_Legs_Only section of the file we specify the callback key data. The first property we must set is called CallbackCount, which specifies the number of callback keys we would like to register with this animation set. The ‘Walk_Legs_Only’ animation set is a short looping walking sequence which walks the legs in a ‘Left-Right-Left-Right’ motion. That is, each foot makes contact with the ground twice. Since we wish to trigger a sound whenever a foot touches the ground, this means we will need to register four callback keys to play four footstep sounds at different times.

Following the CallbackCount property is the actual callback key definitions list. Each property should use the format Callback[*N*].*Value* where *N* is the index of the callback key that we are defining for that particular animation set. That is, the indices are local to the section. *Value* should be replaced with the name of the property we are defining.

Each callback key should have at least two properties in common regardless of what type it is. It should have a Ticks property which describes where in the periodic timeline of the animation set this key should be triggered and an Action property which describes what type of callback key definition it is. Currently, we have only defined one type of callback key definition called PLAY_SOUND, but you can define you own custom callback key types and develop the objects to parse and handle them.

A PLAY_SOUND callback definition encapsulates the registration of a sound event with an animation set. However, we can assign multiple sound effects to this callback key. For example, for each of our callback keys we have defined four wave files to be played. This allows us to either cycle through this sound list each time the callback key is triggered or we can allow our sound playing object to choose one from the list at random. You will notice that for our first callback key definition we specify the names of four wave files. These represent four slightly different sound effects of a foot hitting the ground. We also set the play mode to RANDOM so that we know we can just choose any wave file from the list when this callback key is triggered. You will notice that each of the four keyframes is set up the same way. Having multiple sound effects for a single key really helps in this case. If we had just a single footstep sound it would be very repetitive and would produce a 'Thud Thud Thud Thud' sound as the player walks. Because each key when triggered will be choosing a footstep sound to play at random we get a non-repetitive footsteps sound like 'Thud Crunch Stomp Thud Stomp Crunch Crunch'. The end result is much more pleasing because it sounds like the player is walking over an uneven terrain with many different materials underfoot. The other option for this value is CYCLE, which means we wish to step through the list in order every time the callback key is triggered (with wraparound back to the top of the list when all sounds have been played).

As you can see, if the callback key definition is of type PLAY_SOUND then the PlayMode, SampleCount, and SampleList properties are also expected to be defined. If you wish to register a single sample then just set sample count to 1 and list only one wave file for that key.

Now that we know how our data is laid out in the .cbd file, let us have a look at the methods of the IActionData interface and discuss there parameter lists.

The IActionData Interface

```
virtual HRESULT CreateCallback(          ULONG Index,
                                         LPVOID pContext,
                                         LPCTSTR strActorFile,
                                         void * pCallbackData,
                                         LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                         D3DXKEY_CALLBACK * pKey ) = 0;
```

The IActionData interface is the base class for objects which encapsulate callback key actions (e.g., playing a sound). The CreateCallback function will be called to extract the information for a given key as each IActionData derived object encapsulates the data for a single callback key. That is, it is an object of this type that will have its pointer stored along with the callback key in the animation set and will contain the information that must be triggered when the callback is executed.

The CreateCallback method will be called by the CScene::CollectCallbacks method whenever it wishes to create a new callback key. It will create a new IActionData derived object and call this method for a given animation set, passing in the index of the callback key. This index is used by our derived object to know which callback key we should extract the information from the .cbd file for. If Index was set to 2, the IActionData object should extract and encapsulate the information for the second callback key for the animation set passed (as the fifth parameter). As the second parameter, context data can also be

passed. This is always a pointer to the CScene object in our demo applications. As the third parameter the name of the actor is passed and as the fourth parameter the name of the .cbd file is passed (in our example, but this can be whatever context data was passed into the LoadActorFromX function). As the fifth parameter we pass in the animation set whose callback key we wish to create. The final parameter is a pointer to a D3DXKEY_CALLBACK structure that will eventually store a pointer to this IActionData derived object before being registered with the set.

```
virtual HRESULT HandleCallback( UINT Track, CActor * pActor ) = 0;
```

The HandleCallback function is not called during callback registration (inside CScene::CollectCallbacks); it is called inside the callback handler function when the callback key is triggered. The callback handler will be passed a pointer to the IActionData derived object. The object is expected to know what it has to do. For example, if our object is a sound player object, then it should know that when its HandleCallback function is called, it should play that sound without having to burden the CScene object with the responsibility (as we did previously). We will see this function being used later when we look at the code to the callback handler.

In our project we derive a class called CSoundCallback from IActionData. It is used to encapsulate a single callback key's sound data. In our example, that means we will create four callback keys for our Walk_Legs_Only animation set and each will store the address of a CSoundCallback object as its context pointer. We will look at the code to this object in a moment, but for now just know that its CreateCallback method will extract and load the sound data from the .cbd file for the given callback key that it represents and its HandleCallback function will know how to pick a random sound from the list and play it.

At this point, we can get some clarity by looking at the new implementation of the CScene::CollectCallbacks function. Remember, this function is called from an actor just after its animation data has been loaded. It will be called once for each animation set, allowing this function a chance to add some callback keys to it. The final parameter to this function is a D3DXKEY_CALLBACK array that was allocated by the actor. You simply store any callback keys you would like to register in this array and on function return, the actor will clone the animation set and add the callback keys you specified to the set.

```
ULONG CScene::CollectCallbacks(  LPVOID pContext,
                                 LPCTSTR strActorFile,
                                 void * pCallbackData,
                                 LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                 D3DXKEY_CALLBACK pKeys[] )
{
    IActionData * pData  = NULL;
    LPTSTR strDefinition = (LPTSTR)pCallbackData;
    LPCTSTR strSetName   = pAnimSet->GetName();
    ULONG CallbackCount, i, KeyCount = 0;
    TCHAR strBuffer[128], strKeyName[128];

    // Retrieve the number of callbacks defined
    CallbackCount = GetPrivateProfileInt( strSetName, _T("CallbackCount"),
                                          0,
                                          strDefinition );
```

```
    if ( CallbackCount == 0 ) return 0;
```

The first thing this function does is cast the pCallbackData to a string as this will contain the name of the .cbd file that was passed by the application into the CActor::LoadActorFromX function. It is stored in the strDefinition local variable. It then fetches the name of the passed animation set so that it can be determined whether this function is currently being called for a set we have any interest in registering callback keys for. We then try and retrieve the CallbackCount property for the section in the file that matches the name of the current animation set we are processing. If there is no definition for this animation set then zero will be returned by default and we can exit.

At this point, if CallbackCount is non-zero it means there was a section defined in the file for the current animation set we are processing.

In the next section of code we loop through the number of callback key definitions contained in the .cbd file for the current animation set we are processing. For each one, we first grab the value of its Ticks property. Since this is a float, we have to read it in as string and then use sscanf to convert. Notice how we store the extracted periodic position (Ticks) in the Time member of the current element in the passed D3DXKEY_CALLBACK array. KeyCount is a local variable that was initialized to zero at the top of the function and is incremented every time we fill in a new structure in this array.

```
    // Loop through each of the callbacks specified
    for ( i = 0; i < CallbackCount; ++i )
    {
        // Retrieve the 'time' for this particular callback
        _stprintf( strKeyName, _T("Callback[%i].Ticks"), i );

        GetPrivateProfileString( strSetName,
                                 strKeyName, _T("0.0"),
                                 strBuffer,
                                 127,
                                 strDefinition );

        // Scan directly into key time
        sscanf( strBuffer, _T("%g"), &pKeys[ KeyCount ].Time );
```

The next bit is interesting. We extract the Action property of the current callback key definition which allows us to determine what type of callback key this represents. As we currently only support keys of type PLAY_SOUND, if it is not of the correct type we skip it. However, if it is a PLAY_SOUND key definition, we create a new CSoundCallback object.

```
        // Retrieve the 'action' type for this particular callback
        _stprintf( strKeyName, _T("Callback[%i].Action"), i );
        GetPrivateProfileString(    strSetName,
                                    strKeyName,
                                    _T(""),
                                    strBuffer,
                                    127,
                                    strDefinition );

        // What type of callback is this
```

```
        if ( _tcsicmp( strBuffer, _T("PLAY_SOUND") ) == 0 )
            pData = new CSoundCallback( GetGameApp()->GetSoundManager() );
        else
            continue;

        // Validate
        if ( !pData ) continue;
```

We have not looked at the code to the CSoundCallback class yet, but this is our object derived from IActionData that will encapsulate the loading and playing of sound keys. Note that as the sole parameter to its constructor we pass in a pointer to the application's CSoundManager. The CSoundCallback class will need access to the sound manager so that it can create and play CSound objects.

At this point, we have created the new CSoundCallback object for the current key, so we next call its CreateCallback function. This function will extract all the sound information from the .cbd file for the current key and create CSound objects for each sound. They will be stored in a global array of sounds that can be played for this key. Notice that we pass in loop variable 'i' so that it knows the index of the key it needs to extract the data for. If 'i' was set to 2, it would know that it must extract the properties of the second key definition in the file for the current animation set being processed. Also note that the final parameter is the address of the callback key element in the passed array where this object will eventually have its pointer stored. We do not use this parameter in our CSoundCallback::CreateCallback function, but it is useful to have access to it for any custom objects you might develop later.

```
        // Allow the callback to initialize itself
        if (FAILED(pData->CreateCallback( i,
                                          pContext,
                                          strActorFile,
                                          pCallbackData,
                                          pAnimSet,
                                          &pKeys[ KeyCount ] ) ) )
        {
            delete pData;
            continue;

        } // End if failed to create callback
```

At this point the CSoundCallback object will have been created for the current callback key being processed. It will contain an array of CSound objects which can be played at random when the key is triggered.

All that is left to do at the bottom of the loop is store a pointer to this CSoundCallback object in the callback key array and increase the callback key count. When this callback key is triggered, the callback handler will be passed a pointer to this CSoundCallback object. The handler can then issue a call to the CSoundCallback::HandleCallback function which will instruct the object to select a CSound object at random from its internal array and play it.

```
        // Store in the key's data area
        pKeys[KeyCount].pCallbackData = (LPVOID)pData;

        // We've successfully added a key
```

```
        KeyCount++;

    } // Next Callback

    // Return the total number of keys created
    return KeyCount;
}
```

We have seen how simple the CScene::CollectCallbacks function is. Most of the heavy lifting has been placed inside the CSoundCallback object so let us have a look at the code to this object next.

The CSoundCallback Class

The CSoundCallback class encapsulates all the data for a single PLAY_SOUND callback key. It is defined in CScene.h as shown below. We have snipped the list of functions from the listing to compact listing size. We already know the five methods it exposes: AddRef, Release, and QueryInterface inherited from IUnknown and CreateCallback and HandleCallback inherited from IActionData. Here we see the list of its member variables and structures followed by a description of each.

Excerpt from CScene.h

```
class CSoundCallback : public IActionData
{
public:

    enum PlayStyle { PLAY_LOOPING = 0, PLAY_ONCE = 1 };

    … Snip… (Functions)

private:
    struct SampleItem
    {
        TCHAR   FileName[512];          // Filename for the loaded sample
        CSound *pSound;                 // The actual sound object.
    };

    ULONG           m_nRefCount;        // Reference counter
    bool            m_bRandom;          // We should pick a random sample
    ULONG           m_nSampleCount;     // Number of samples stored
    ULONG           m_nNextPlayIndex;   // Which sample did we last play?
    SampleItem    * m_pSamples;         // The array which stores all the samples loaded.
    CSoundManager * m_pSoundManager;    // The sound manager specified by our application
};
```

ULONG m_nRefCount

This variable holds the current reference count of the object. Our CSoundCallback object must implement the same reference counting mechanism as a COM object so that the actor can safely release this object using the IUnknown::Release method. This is important because, although a pointer to this object will be stored in the callback key, it will be stored as a void pointer and the actor has no idea what sort of object it is or how to delete it. As long as the actor can rely on the fact that the object stored in each callback key is derived from IUnknown, it can cast the void pointer to an IUnknown pointer and instruct the object to destroy itself.

bool m_bRandom
This boolean will be set to true if this CSoundCallback object has been placed into random playlist mode. When in random mode, every time this object's HandleCallback method is called it will randomly select a sample to play from its internal array. We saw when we examined the contents of the .cbd file that each of our four keys were set to random mode. Therefore, the four CSoundCallback objects that we create will each have this variable set to true.

ULONG m_nSampleCount
This contains the number of sampled sounds this object stores in its internal sound list. It describes the number of SampleItem structures in the array discussed below. For each of the CSoundCallback objects we create in our demo this value will be set to 4 because each callback key definition in the .cbd file has four wave file sounds associated with it.

SampleItem * m_pSamples
This is an array of SampleItem structures. Each element in the array contains the name of the sound file that is stored there and a pointer to the CSound object that was created for that sound.

```
struct SampleItem
{
    TCHAR   FileName[512];    // Filename for the loaded sample
    CSound *pSound;           // The actual sound object.
};
```

In a moment we will take a look at the CSoundCallback::CreateCallback function, which is called just after the object is instantiated in CScene::CollectCallbacks (shown earlier). This function extracts the wave file names from the passed .cbd file and uses the application's CSoundManager object (passed in the constructor) to create a CSound object for each wave file listed. These pointers are then stored in this array along with the original name of the sound file (taken from the .cbd file). It is this list that we choose a sound effect to play from each time the key is triggered and the object's HandleCallback function is called. We will see this array having its contents created when we look at the CreateCallback method in a moment.

ULONG m_nNextPlayIndex
This value is used only if the object is placed into CYCLE mode. It stores the index of the current position in the sound list so we know which sound to play next when the HandleCallback method is called. This value will start of at zero and be incremented every time the HandleCallback method is called and a sound is played. When this value reaches the end of the list it will be wrapped back around to zero so we start playing sounds from the start of the list again.

CSoundManager *m_pSoundManager
In this member we store a pointer to the CSoundManager object which will be used to create CSound objects. This pointer must be passed into the constructor as we saw earlier when the object was being created in the CScene::CollectCallbacks function (we passed in the CGameApp's CSoundManager object pointer).

Let us now look at the methods of the CSoundCallback object, which are all very simple.

CSoundCallback::CSoundCallback

We already saw the constructor being called from the CScene::CollectCallbacks function. It is passed the application's CSoundManager pointer which it stores in a member variable. It also switches the object's playlist into CYCLE mode (random=false) by default and sets the sample array pointer to NULL. We also set the initial reference count of the object to 1.

```
CSoundCallback::CSoundCallback( CSoundManager * pSoundManager )
{
    // Setup default values
    m_pSoundManager     = pSoundManager;
    m_bRandom           = false;
    m_pSamples          = NULL;
    m_nSampleCount      = 0;
    m_nNextPlayIndex    = 0;

    // ************************************
    // *** VERY IMPORTANT
    // ************************************
    // Set our initial reference count to 1.
    m_nRefCount = 1;

}
```

Notice how we also set the play index to zero so that the first sound to be played in cycling mode will be the first one loaded into the sample array.

CSoundCallback::~CSoundCallback

The destructor is also very simple. Its main task is to release its array of SampleItem structures if it exists. Keep in mind that each structure in this array will store a pointer to a CSound object, so we must release that too.

```
CSoundCallback::~CSoundCallback( )
{
    ULONG i;

    // Release any memory
    if ( m_pSamples )
    {
        for ( i = 0; i < m_nSampleCount; ++i )
        {
            // Delete the sound object if it exists.
            if ( m_pSamples[i].pSound ) delete m_pSamples[i].pSound;
        } // Next Sample

        delete []m_pSamples;

    } // End if samples loaded

    // Clear variables
    m_bRandom           = false;
```

```
    m_pSamples          = NULL;
    m_nSampleCount      = 0;
    m_nNextPlayIndex    = 0;
}
```

CSoundCallback::CreateCallback

This is the function where all of our object initialization happens. It is called from the CScene::CollectCallbacks function when it is determined that there is a callback key definition in the .cbd file for the current animation set being processed. When this function is called it is passed the index of the callback key inside the .cbd file that this object is to encapsulate. For example, if the Index parameter passed was 4 and the fifth parameter was a pointer to an animation set with the name 'Walk_Legs_Only', that would mean this object is to fetch the data for the fourth callback key in the .cbd section called [Walk_Legs_Only]. The function is also passed the callback function context data. This is an arbitrary data pointer that is registered with the actor at the same time the callback function itself is registered. We always store the CScene pointer along with the callback function so that the static CScene::CollectCallbacks function can access the non-static members of the object instance. The function is also passed a string containing the name of the actor that is having the callback keys registered and an additional context data pointer called pCallbackData. This is the pointer we send into the LoadActorFromX function which (in our application) contains the name of the .cbd file that we want the CollectCallbacks function to use. A pointer to the interface of the animation set currently requesting callback key registration is also passed. The final parameter is a pointer to the D3DXKEY_CALLBACK structure that will eventually store a pointer to this CSoundCallback object.

This function must extract the data from the .cbd file and create a new CSound object for each sound file listed for this key. These CSound objects are then stored in the SampleItem array so that they can be played when the key is triggered by the animation system. Here is the first section of code.

```
HRESULT CSoundCallback::CreateCallback(   ULONG Index,
                                          LPVOID pContext,
                                          LPCTSTR strActorFile,
                                          void * pCallbackData,
                                          LPD3DXKEYFRAMEDANIMATIONSET pAnimSet,
                                          D3DXKEY_CALLBACK * pKey )
{
    LPTSTR strDefinition = (LPTSTR)pCallbackData;
    LPCTSTR strSetName   = pAnimSet->GetName();
    TCHAR strBuffer[128], strKeyName[128];
    ULONG i;

    // Random or cycle sample playlist?
    _stprintf( strKeyName, _T("Callback[%i].PlayMode"), Index );

    GetPrivateProfileString(  strSetName,
                              strKeyName,
                              _T("CYCLE"),
                              strBuffer,
                              127,
                              strDefinition );
```

```
    if ( _tcsicmp( strBuffer, _T("RANDOM") ) == 0 ) m_bRandom = true;
```

Notice that we store the name of the .cbd file (pCallbackData) in the strDefinition local variable and store the name of the animation set in the strSetName local variable. We then fetch the play mode for the current key. The name of the key is built by substituting the Index parameter into the string "Callback[Index].Playmode". Remember, Index is the local index of the key we are fetching for a given animation set. We then use the GetPrivateProfileString method to fetch this property. The parameters specify that we want to retrieve a property from a section with a name that matches the name of the animation set, with a key name of 'Callback[*n*].PlayMode' where *n* is the current index of the callback key. If the property is not found, a default string of "CYCLE" will be returned, placing the object into non-random mode. As the final parameter we pass in strDefinition, which now contains the name of the .cbd file we wish to extract the data from.

When the function returns, the strBuffer array should contain the value of this property (either RANDOM or CYCLE). If it contains the word "RANDOM", then the m_bRandom boolean property of the object is set to true, placing the object into random playlist mode.

In the next section we fetch the sample count of the current key and allocate the object's array of SampleItem structures to hold that many samples.

```
    // Retrieve the number of samples we're using
    _stprintf( strKeyName, _T("Callback[%i].SampleCount"), Index );

    m_nSampleCount=GetPrivateProfileInt(strSetName, strKeyName, 0,strDefinition );
    if ( m_nSampleCount == 0 ) return E_INVALIDARG;

    // Allocate enough space for the sample data
    m_pSamples = new SampleItem[ m_nSampleCount ];
    if ( !m_pSamples ) return E_OUTOFMEMORY;

    // Clear out array.
    ZeroMemory( m_pSamples, m_nSampleCount * sizeof( SampleItem ) );
```

Now we will set up a loop that will fetch the name of each sample from the file and will store it in the SampleItem array. We will then pass this filename to the CSoundManager::Create method so that it can create a CSound object for the respective sound file. Notice how the m_pSamples[i].pSound pointer is passed as the first parameter to the Create method so that on function return, the new CSound object created will have its pointer stored in the correct place in the sample array.

```
    // Loop through and retrieve the items
    for ( i = 0; i < m_nSampleCount; ++i )
    {
        // Retrieve sample filename
        _stprintf( strKeyName, _T("Callback[%i].Sample[%i].File"), Index, i );

       GetPrivateProfileString(     strSetName,
                                    strKeyName,
                                    _T(""),
                                    m_pSamples[i].FileName,
```

```
                                    511,
                                    strDefinition );

        // Attempt to load the sound
        m_pSoundManager->Create( &m_pSamples[i].pSound, m_pSamples[i].FileName );

    } // Next Sample

    // Success!
    return S_OK;
}
```

So far we have covered all the steps involved in storing our CSoundCallback pointers in the callback keys for a given animation set. In summary:

1. The COLLECTKEYS callback function is registered with actor prior to load call.
2. Call to LoadActorFromX is passed a context pointer that will be passed to the callback function registered in step 1.
3. After the actor has loaded the animation data it will loop through each loaded animation set and call the callback function, passing it the context data pointer passed to the actor in step 2. We use this to pass a .cbd filename to the function (in our lab project).
4. The callback function is called by the actor to provide any callback keys for the current animation set being processed. This function opens the .cbd file to see if there is a section specific to the current animation for which the callback function in being invoked. If a section exists with the same name as the animation set, it knows there must be callback keys to set up.
5. For each key definition for a given animation set, a CSoundCallback object is created and the CSoundCallback::Create method is called to build its internal playlist of CSound objects.
6. The CSoundCallback object pointer, along with the Ticks value of that key (extracted from the cbd file), is stored in a D3DXKEY_CALLBACK structure and placed in the callback key array that is returned to the actor.
7. The actor will clone the animation set to create a new one with the desired keys.

At this point then, our actor will be fully created and each animation set that we desire will have their respective callback keys defined. The callback keys themselves store a pointer to a CSoundCallback object which is passed to the callback handler function when the key is triggered. The callback handler function can then simply call the CSoundCallback::HandleCallback method which will instruct it to pick and play a new sound. Let us have a look at the HandleCallback method next.

CSoundCallback::HandleCallback

This function has a very simple job. It will be called by the callback handling function every time the callback key is triggered. It has to decide which sound to play from its play list (based on its mode of operation) and play that sound.

The Handle Callback function accepts two parameters, although this particular object's implementation of the function uses neither of them. However, the parameter list laid out in the base interface dictates that this function should be passed the index of the track on the controller from which the callback key is

being triggered and a pointer to the actor who is currently using that controller. Although this function is very simple and has no need for these parameters, you can certainly imagine how your own callback key objects (derived from IActionData) may well need them. For example, you might use the track index to fetch the weight of the track that triggered the key. Perhaps a sound is only played if the weight is high enough. We might imagine for example that we want to trigger a sound for the arms when moving up into the gun position (a gun reload sound perhaps). However, the shooting pose is a single frame set and as such, we can not just register a sound with it as it will repeat for as long as the character is holding his gun to his shoulder. One way around this would be to fetch the weight of the track and only trigger the reload sound if the weight was between say 0.5 and 0.6. This would mean his arms are roughly 50% of the way through the transition into the shooting pose. Although this is not a great example, it demonstrates why having the track index can be useful. Of course, having the track by itself is not useful at all if we have no idea which animation controller this track index is related to. Therefore, a pointer to an actor is passed so that the function can retrieve its controller's track properties.

In our case the function is simple -- we just wish to play a sound. If the m_bRandom boolean variable is set to true then we generate a random index into the SampleItem array. If not, then we are in cycle mode and the m_nNextPlayIndex variable will hold the index of the next sound to play (0 initially). We use our index to access the SampleItem array and call the Play method of the CSound object that is stored there. Here is all of the code:

```
HRESULT CSoundCallback::HandleCallback( UINT Track, CActor * pActor )
{
    ULONG PlayIndex;

    // Select item from 'play list'
    if ( m_bRandom )
        PlayIndex = rand() % m_nSampleCount;
    else
        PlayIndex = m_nNextPlayIndex;

    // Play the sample if it's valid
    if ( m_pSamples[PlayIndex].pSound ) m_pSamples[PlayIndex].pSound->Play( );

    // Cycle the next play index
    m_nNextPlayIndex++;
    m_nNextPlayIndex %= m_nSampleCount;

    // Success!
    return S_OK;
}
```

Notice that after the sound has been played, the m_nNextPlayIndex variable is incremented and wrapped around to zero if it exceeds the maximum sample count.

That covers the code for the CSoundCallback object. This class also has AddRef, Release, and QueryInterface methods, but you should be very comfortable with these types of functions by now, so will not show or discuss their implementation again here.

Updating CActionHandler

In Chapter Ten we learned how to use the callback handler features of the D3DXAnimationController. In short, we had to derive a class from the ID3DXAnimationCallbackHandler abstract base class. This base interface specified a single method that must be implemented in the derived class, called HandleCallback. We pass an instance of this object to the AdvanceTime method and the HandleCallback method will be invoked whenever a callback key is triggered. Nothing much has changed in this project except that now our CActionHandler class has an added CActor member pointer and an extra method to set it. Here is our updated CActionHandler object:

Except from CScene.h

```
class CActionHandler : public ID3DXAnimationCallbackHandler
{
public:
    CActionHandler( );

    STDMETHOD(HandleCallback)(THIS_ UINT Track, LPVOID pCallbackData);


    // CActionHandler
    void SetCurrentActor( CActor * pActor );

private:
    CActor * m_pActor;  // Storage for the actor that is currently being advanced
};
```

Why have we added the actor pointer? As we discussed earlier, when a callback key is triggered, the CActionHandler::HandleCallback method will be called by the controller and passed the index of the track that triggered the event. We are also passed a void pointer to the individual key's context data (which in our case will be a pointer to the CSoundCallback object for that key). This function can then simply call the CSoundCallback object's HandleCallback method because, as we have seen, it is this method which takes care of actually playing the sound. Below we show our implementation of the CActionHandler::HandleCallback function. Remember, it is this method that is called by the D3DXAnimationController when its parent object is passed into the AdvanceTime call. If we take a look at the HandleCallback method of this object it will become clear why that actor pointer is needed.

CActionHandler::HandleCallback

Below we see all of the code to the handling function that is called by the controller when a callback key is triggered. It simply casts the passed data pointer to an IUnknown pointer and then uses its QueryInterface method to make sure it is a pointer to an IActionData derived object. Once we have an IActionData pointer, we can call its HandleCallback method. In the case of our CSoundCallback object, it will choose a sound to play from its list of sounds. Notice however that the IActionData::HandleCallback function expects to be passed a pointer to an actor; information that is not provided by the animation controller. We need the current actor being updated stored in the handler

object so that the track ID we are passed actually means something. Without us knowing what controller spawned the callback, the track index is useless to us.

```
HRESULT CActionHandler::HandleCallback( UINT Track, LPVOID pCallbackData )
{
    IUnknown    * pData   = (IUnknown*)pCallbackData;
    IActionData * pAction = NULL;

    // Determine if this is in a format we're aware of
    if ( FAILED( pData->QueryInterface( IID_IActionData, (void**)&pAction )))
        return D3D_OK; // Just return unhandled.

    // This is our baby, lets handle it
    pAction->HandleCallback( Track, m_pActor );

    // Release the action we queried.
    pAction->Release();

    // We're all done
    return D3D_OK;
}
```

So, we need to place an additional function call in our CScene::AnimateObjects function. Prior to calling AdvanceTime and passing in the handler object, it will call the handler object's SetCurrentActor method. Below we show the CScene::AnimateObjects method with this additional function call highlighted in bold.

```
void CScene::AnimateObjects( CTimer & Timer )
{
    ULONG i;
    static CActionHandler Handler;

    // Process each object for coll det
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CObject * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Get the actor pointer
        CActor * pActor = pObject->m_pActor;
        if ( !pActor ) continue;
        if ( pObject->m_pAnimController )
             pActor->AttachController( pObject->m_pAnimController,
                                       false,
                                       pObject->m_pActionStatus );

        // Set this actor in the handler so the callbacks have access to it
        Handler.SetCurrentActor( pActor );

        // Advance time
        pActor->AdvanceTime( Timer.GetTimeElapsed(), false, &Handler );

    } // Next Object
}
```

As you can see, we create a handler object on the stack at the top of the function and pass the same object into the AdvanceTime call for each actor. Therefore, before we call AdvanceTime, we set the handler's actor pointer to the current actor we are processing so that when the callback handler is called by the controller, it has access to the actor from which it was triggered. The callback function then has the ability to query the actor to find out the state of its controller and the track index passed in becomes useful.

That concludes our coverage of the updated callback key system. Hopefully you will find it relatively straightforward to use. In the absence of a proper application sound manager, it provides us a fair bit of flexibility for such a small amount of effort.

Application Framework Upgrades

We are approximately halfway through the course at this point. Moving forward we will spend the remainder of the course discussing topics less related to the DirectX API and more related to general games programming. So before we do that, we will use Lab Project 12.2 to make some changes to existing classes to make our framework more usable in future lessons. We will also upgrade our framework's IWF code to provide more comprehensive support for the various entity types exported from GILES™. This will allow us to consolidate many of the topics and techniques we have learned about so far in the entire series. Our application framework will soon support heightmap based terrains, actors, and static scenes, all within our CScene class. We will also add support for the loading of fog parameters from IWF files and the rendering of the scene using those fog parameters. From here on in, our application framework should be able to load and render most IWF files exported from GILES™ that we care to throw at it, even if that IWF file contains multiple terrain entities, external references to animated X files, and static geometry.

A Modular Approach to Querying Supported Texture Formats

In previous applications our CGameApp class would query the support of texture formats that its various components would need. The querying was done in CGameApp::TestDeviceCaps which was called at application startup from the CGameApp::InitInstance method. For example, if the terrain class needed an alpha texture format for its water plane, the test for a suitable alpha format was performed by CGameApp and the resulting format was passed to the terrain object for storage. It could then create textures using this format when the terrain was being initialized. Likewise, if the scene required support for compressed textures, the CGameApp::TestDeviceCaps function would have to be modified to test for a suitable compressed texture format, which would then be passed to the scene object for storage and texture creation during scene initialization. This approach served us fairly well when we are only plugging a few components into our framework, but as we start plugging in more components which may each have different texture needs, it becomes cumbersome.

Rather than having to alter the code to our CGameApp class every time we plug in a new component that requires a texture format we have not previously provided support for, it would be much nicer if the component itself could intelligently chose which format it wants to use to complete its task. However,

we do not want each component to have to duplicate the device tests that are done in the CGameApp::TestDeviceCaps method.

The solution we provide in this lab project, which will be used in all future projects, introduces a new class called CTextureFormatEnum. This class contains a list of all texture formats that are supported by DirectX paired with a boolean value indicating whether or not this texture format is supported by the current device.

The CGameApp object will contain a member of this type and in the TestDeviceCaps method it will now simply call the CTextureFormatEnum::Enumerate method. This method will loop through every DirectX supported format and test for device compliance. When this function returns, the CTextureFormatEnum will contain a 'true' or 'false' property paired with each possible texture format.

There is nothing too clever about this, but what this new class also has are methods that allow any object to say things like "Give me the best compressed texture format with an alpha channel". The object will have logic that will loop though each supported format and return the best one that matches the criteria.

Rather than each component that we plug into framework (CScene, CTerrain etc) having to be passed supported texture formats by the CGameApp class, the CGameApp object will essentially just call this object's Enumerate method to allow us to find which formats are supported. Each component we plug in to our framework can then simply be passed a pointer to this CTextureFormatEnum object so that the component can query its method to retrieve the formats it desires. The CGameApp class no longer has to be continually edited and extended to support more formats every time we plug in a new component with different texture needs. Instead, we just make sure that CGameApp calls the SetTextureFormat method for the component and passes in a pointer to this texture format enumeration object. We will have to make sure that all the components (e.g., CTerrain, CScene, etc.) have a function of this type and the ability to store a pointer to the enumeration object.

If you look at the CGameApp class, it now has a new member as shown below:

Excerpt from CGameApp.h

```
CTextureFormatEnum      m_TextureFormats;
```

This new member is an instance of the type of the new class we are about to create. Looking inside the CGameApp::TestDeviceCaps method we can see that it no longer performs any texture format queries on the device; instead it simply calls the Enumerate method of this new object.

Excerpt From CGameApp::TestDeviceCaps

```
ULONG      Ordinal = pSettings->AdapterOrdinal;
D3DDEVTYPE Type    = pSettings->DeviceType;
D3DFORMAT  AFormat = pSettings->DisplayMode.Format;

// Enumerate texture formats the easy way :)
if ( !m_TextureFormats.Enumerate( m_pD3D, Ordinal, Type, AFormat )) return false;
```

We will cover all the code to the CTextureFormatEnum object in a moment. For now, just know that by calling its Enumerate method and passing in the current device, the adapter ordinal, the type of device

(HAL for example) and the current adapter format, the object will have its internal data filled with a list of supported formats.

At this point, any component with a pointer to this object can use its methods to ask for supported formats. The following code snippet is taken from the CGameApp::BuildObjects function and shows the pointer to this object being passed to the CScene object for storage.

```
m_Scene.SetTextureFormat( m_TextureFormats );
m_Scene.LoadSceneFromIWF( m_strLastFile, LightLimit )
```

Notice how we call the SetTextureFormat method before calling the loading function. This allows the scene to use this object to query for supported texture formats during the loading process. What is also nice is that the scene object can pass this pointer to any of its components (e.g., CTerrain objects) so that they too can use the object to query texture formats independently.

Because the scene object now has a pointer to the CTextureFormatEnum object, whenever it needs to create a texture, it can just call the object's GetBestFormat method to return the most desirable format for the current device.

Below we see a line of code from the CScene::CollectTexture function, which we use to load all the scene's textures. In this example, the GetBestFormat method is being used with no parameters. This means it will use the default logic when choosing the best format -- finding the best non-alpha compressed texture format. There will also be a version of this function that allows you to override this logic by specifying request parameters

Excerpt from CScene::CollectTexture

```
D3DXCreateTextureFromFileEx( pScene->m_pD3DDevice,
  Buffer,
  D3DX_DEFAULT, D3DX_DEFAULT,
  3, 0,
  pScene->m_TextureFormats.GetBestFormat(),
  D3DPOOL_MANAGED,
  D3DX_DEFAULT,
  D3DX_DEFAULT,
                              0,
  NULL,
  NULL,
  &pNewTexture->Texture );
```

Notice how easy it is for a component to fetch the best compatible format now.

The CTextureFormatEnum Object

The declaration and code for this class are stored in the files 'CTextureFormatEnum.h' and 'CTextureFormatEnum.cpp' respectively. Below we see the class declaration minus its methods to compact the listing. We will discuss its methods and the code they contain in a moment.

Excerpt from CTextureFormatEnum.h

```
typedef std::map<D3DFORMAT, bool> FormatEnumMap;

class CTextureFormatEnum
{
public:
    …
    …
    // Methods not Shown here
    …
    …
    FormatEnumMap   m_TextureFormats;       // Formats
    bool            m_bPreferCompressed;    // Compressed Texture Bias Mode
    bool            m_bAlpha;               // Should we query alpha formats only?
};
```

The class has three member variables. The last two are simply booleans that contain the default mode of the query methods (discussed in a moment). That is, if m_bAlpha is set to true then when we ask the object to return the best supported format, it will prefer one with alpha if it exists and return that. Likewise, if the m_bPreferCompressed boolean is set to true, the query methods will favor compressed formats. The states of these booleans can be set and retrieved via simple member functions. They can also be overridden by passing parameters into the query functions. Having these defaults just provides a nice way for the application to set the most common parameters that will be required so that it can call the parameterless version of GetBestFormat when it wants to fetch a compatible texture format.

The top-most member is of type FormatEnumMap. This is a typedef for an STL map object that stores a D3DFORMAT and a boolean in key/value pairs. For those of you a little rusty with maps, a map is a way for us to essentially build a table of key-value pairs (like a dictionary). We might imagine that an array is like a linear map where the keys are the array indices and the values are stored in those elements. However, in a map, the keys can be anything you like (floats, strings, or even entire structures). Internally, maps are stored as binary trees, which makes the searching for a key/value pair very quick indeed.

So why are we using a map? We are using map because each key will be a D3DFORMAT structure and the value associated with that key will be a Boolean that is set to true or false depending on whether the format is supported. This allows our object, for example, to record support for the X8R8G8B8 texture format as shown below.

```
m_TextureFormats[ D3DFMT_X8R8G8B8 ] = true;
```

With that in mind, let us examine the code to our new CTextureFormatEnum object.

CTextureFormatEnum::CTextureFormatEnum

The constructor is simple. The default behavior is to prefer compressed formats over non-compressed ones and to prefer non-alpha formats unless an alpha format is explicitly requested when the query is performed. In the final step, the constructor calls the Init method of the object which initializes the STL map of texture formats so that nothing is initially supported until the Enumerate method is called.

```
CTextureFormatEnum::CTextureFormatEnum()
{
    // Setup any required values
    m_bPreferCompressed = true;
    m_bAlpha            = false;

    // Initialize the class
    Init();
}
```

CTextureFormatEnum::Init

This function first empties the STL map by calling its Clear method. At this point there will be no key/value pairs in the map. We then add the keys for each possible texture format. We use a default value of false since we have not yet queried any of these formats on the device to test for support.

```
void CTextureFormatEnum::Init()
{
    // First clear the internal map if it's already been used.
    m_TextureFormats.clear();

    // Setup each of the map elements
    m_TextureFormats[ D3DFMT_R8G8B8        ] = false;
    m_TextureFormats[ D3DFMT_A8R8G8B8      ] = false;
    m_TextureFormats[ D3DFMT_X8R8G8B8      ] = false;
    m_TextureFormats[ D3DFMT_R5G6B5        ] = false;
    m_TextureFormats[ D3DFMT_X1R5G5B5      ] = false;
    m_TextureFormats[ D3DFMT_A1R5G5B5      ] = false;
    m_TextureFormats[ D3DFMT_A4R4G4B4      ] = false;
    m_TextureFormats[ D3DFMT_R3G3B2        ] = false;
    m_TextureFormats[ D3DFMT_A8            ] = false;
    m_TextureFormats[ D3DFMT_A8R3G3B2      ] = false;
    m_TextureFormats[ D3DFMT_X4R4G4B4      ] = false;
    m_TextureFormats[ D3DFMT_A2B10G10R10   ] = false;
    m_TextureFormats[ D3DFMT_A8B8G8R8      ] = false;
    m_TextureFormats[ D3DFMT_X8B8G8R8      ] = false;
    m_TextureFormats[ D3DFMT_G16R16        ] = false;
    m_TextureFormats[ D3DFMT_A2R10G10B10   ] = false;
    m_TextureFormats[ D3DFMT_A16B16G16R16  ] = false;
    m_TextureFormats[ D3DFMT_A8P8          ] = false;
    m_TextureFormats[ D3DFMT_P8            ] = false;
    m_TextureFormats[ D3DFMT_L8            ] = false;
    m_TextureFormats[ D3DFMT_A8L8          ] = false;
    m_TextureFormats[ D3DFMT_A4L4          ] = false;
```

```
    m_TextureFormats[ D3DFMT_V8U8           ] = false;
    m_TextureFormats[ D3DFMT_L6V5U5         ] = false;
    m_TextureFormats[ D3DFMT_X8L8V8U8       ] = false;
    m_TextureFormats[ D3DFMT_Q8W8V8U8       ] = false;
    m_TextureFormats[ D3DFMT_V16U16         ] = false;
    m_TextureFormats[ D3DFMT_A2W10V10U10    ] = false;
    m_TextureFormats[ D3DFMT_UYVY           ] = false;
    m_TextureFormats[ D3DFMT_R8G8_B8G8      ] = false;
    m_TextureFormats[ D3DFMT_YUY2           ] = false;
    m_TextureFormats[ D3DFMT_G8R8_G8B8      ] = false;
    m_TextureFormats[ D3DFMT_DXT1           ] = false;
    m_TextureFormats[ D3DFMT_DXT2           ] = false;
    m_TextureFormats[ D3DFMT_DXT3           ] = false;
    m_TextureFormats[ D3DFMT_DXT4           ] = false;
    m_TextureFormats[ D3DFMT_DXT5           ] = false;
    m_TextureFormats[ D3DFMT_L16            ] = false;
    m_TextureFormats[ D3DFMT_Q16W16V16U16   ] = false;
    m_TextureFormats[ D3DFMT_MULTI2_ARGB8   ] = false;
    m_TextureFormats[ D3DFMT_R16F           ] = false;
    m_TextureFormats[ D3DFMT_G16R16F        ] = false;
    m_TextureFormats[ D3DFMT_A16B16G16R16F  ] = false;
    m_TextureFormats[ D3DFMT_R32F           ] = false;
    m_TextureFormats[ D3DFMT_G32R32F        ] = false;
    m_TextureFormats[ D3DFMT_A32B32G32R32F  ] = false;
    m_TextureFormats[ D3DFMT_CxV8U8         ] = false;

}
```

This function really shows us what the object stores -- a big list of possible texture formats with a boolean describing its support status on the device. Of course, this is all changed when the Enumerate method is executed (usually the first method our application will call).

CTextureFormatEnum::Enumerate

This function is called to update the status of each key in the STL map to reflect the support offered by the device. The function is passed a pointer to the device and the adapter ordinal. It is also passed the type of device we are interested in enumerating formats for (HAL for example) and the current adapter format since this is a factor in determining what texture formats are compatible.

The first thing we do is set up an iterator to begin searching through the key/values pairs in the map. Each key/value pair of a map is accessed by the map iterator object using its ‘first’ and ‘second’ members. That is, as the iterator accesses each row of map data, its ‘first’ member will contain the key (the format itself, such as D3DFMT_X8R8G8B8) and the ‘second’ member can be used to set or retrieve the value associated with that key (the boolean status).

Once we setup the iterator to point at the first row of map data, we will loop through all the rows added in the previous function. We will then use the IDirect3DDevice9::CheckDeviceFormat function to query support for each mode and set the boolean status accordingly. At the end of this function, any modes that are supported in the map will have a ‘true’ value associated with them.

Here is the code. First we use the map's Begin method so the iterator is pointing to the first row of data. We then set up a loop that increments the iterator until the End is reached. We call the CheckDeviceFormat function for the current key and set its value to true if the function succeeds.

```
bool CTextureFormatEnum::Enumerate( LPDIRECT3D9 pD3D,
                                    ULONG Ordinal,
                                    D3DDEVTYPE DeviceType,
                                    D3DFORMAT AdapterFormat )
{
    FormatEnumMap::iterator Iterator = m_TextureFormats.begin();

    // Validate
    if ( !pD3D ) return false;

    // Loop through all texture formats in the list
    for ( ; Iterator != m_TextureFormats.end(); ++Iterator )
    {
        // Test the format
        if ( SUCCEEDED( pD3D->CheckDeviceFormat( Ordinal,
                                                 DeviceType,
                                                 AdapterFormat,
                                                 0,
                                                 D3DRTYPE_TEXTURE,
                                                 Iterator->first ) ) )
        {
            // It's supported, mark it as such
            Iterator->second = true;

        } // End if supported

    } // Next Texture Format Item

    // Success!
    return true;
}
```

CTextureFormatEnum::GetBestFormat

This function is called to retrieve the best supported format. This function (it is overloaded) accepts two parameters that allow the caller to override the default logic of the process. The two booleans are used to specify whether we consider a format with an alpha channel to be a priority and whether compressed formats are preferred over regular formats. This function returns a D3DFORMAT which can then be used to create texture surfaces on the current device.

The first section of the function shown below is the conditional code block that is executed when a compressed format is preferred.

```
D3DFORMAT CTextureFormatEnum::GetBestFormat( bool Alpha, bool PreferCompressed )
{
    // Does the user prefer compressed textures in this case?
    if ( PreferCompressed )
```

```
    {
        // Alpha is required?
        if (Alpha)
        {
            if ( m_TextureFormats[D3DFMT_DXT3] ) return D3DFMT_DXT3;
            if ( m_TextureFormats[D3DFMT_DXT5] ) return D3DFMT_DXT5;

        } // End if alpha required
        else
        {
            if ( m_TextureFormats[D3DFMT_DXT1] ) return D3DFMT_DXT1;

        } // End if no alpha required

    } // End if prefer compressed formats
```

As you can see above, if alpha is preferred we return D3DFMT_DXT3 as the most desirable format if its boolean is set to true in the map. This is a desirable compressed alpha surface because it provides several levels of alpha and it stores the alpha information explicitly. That is to say, each pixel within the texture maintains its own alpha value. If this format is not supported, we return the next best compressed alpha format: D3DFMT_DXT5. This format still provides several levels of alpha, but much of the original alpha data is lost in the compression. Alpha information is interpolated over a wider area of pixels.

Note: There was a detailed discussion of compressed texture formats in Chapter 6 of Module I.

Finally, if alpha is not required then we return D3DFMT_DXT1. This is a compressed format with no support for multiple levels of alpha. This surface only reserves a single bit for alpha information (meaning the pixel is either on or off). This is a most desirable compressed texture format if several levels of alpha are not required since it allows for greater compression (because of the missing alpha information).

That concludes the compressed texture search. The next code block is executed if alpha is required and either non-compressed textures were preferred or a suitable compressed texture could not be found in the above code. In the latter case, even if compressed alpha textures were desired, we will fall back to finding non-compressed alpha formats if the function did not return in the compressed code block.

Our logic is simple -- we are searching for the texture format with the most color and alpha resolution. For alpha requests, we are ideally looking for a 32-bit surface with 8 bits of storage for the red, green, blue, and alpha components. If this is not available, we will make a choice that might seem odd -- we will test support for the A1R5G5B5 format. While this only supports one bit of alpha, as compared to the A4R4G4B4 format which has more bits for alpha and may also be supported, in this latter mode we also only have 4 bits per RGB component. This lack of color resolution generally looks pretty bad. In all situations then, we always give higher priority to the RGB components, even if it means we are returning a texture with only one bit (on/off) of alpha. As the third alpha option we fall back to the A4R4G4B4 format

```
    // Standard formats, and fallback for compression unsupported case
    if ( Alpha )
    {
        if ( m_TextureFormats[ D3DFMT_A8R8G8B8 ] ) return D3DFMT_A8R8G8B8;
        if ( m_TextureFormats[ D3DFMT_A1R5G5B5 ] ) return D3DFMT_A1R5G5B5;
        if ( m_TextureFormats[ D3DFMT_A4R4G4B4 ] ) return D3DFMT_A4R4G4B4;

    } // End if alpha required
```

The next section of code is executed if the caller requested either a non-compressed, non-alpha format, or if a compressed non-alpha format was requested but a solution was not found in the compressed code block. Once again, you can see that we favor resolution for the RGB components in the order that we test and return supported non alpha surfaces.

```
    else
    {
         if ( m_TextureFormats[ D3DFMT_X8R8G8B8 ] ) return D3DFMT_X8R8G8B8;
         if ( m_TextureFormats[ D3DFMT_R8G8B8   ] ) return D3DFMT_R8G8B8;
         if ( m_TextureFormats[ D3DFMT_R5G6B5   ] ) return D3DFMT_R5G6B5;
         if ( m_TextureFormats[ D3DFMT_X1R5G5B5 ] ) return D3DFMT_X1R5G5B5;

    } // End if no alpha required

    // Unsupported format.
    return D3DFMT_UNKNOWN;
}
```

If the bottom line of the function is ever reached, it means we could not find a format that was supported (very unlikely) and we return D3DFMT_UNKNOWN back to the caller. The caller will then know that his request could not be satisfied.

CTextureFormatEnum::GetBestFormat (Overloaded)

The GetBestFormat function is also overloaded so that we can call it with no parameters and use the values of its m_bPreferCompressed and m_bAlpha member variables to control the searching logic. This function simply calls the version of the function we just covered passing in the member variables as the parameters.

This was the version of the function we saw being called in the earlier example of its use.

```
D3DFORMAT CTextureFormatEnum::GetBestFormat( )
{
    return GetBestFormat( m_bAlpha, m_bPreferCompressed );
}
```

Of course, we can also set the m_bAlpha and m_bPreferCompressed booleans via the two simple methods shown below, so that the default logic of the object can be changed.

```
void CTextureFormatEnum::SetCompressedMode( bool Enable )
{
    m_bPreferCompressed = Enable;
}
```

```
void CTextureFormatEnum::SetAlphaMode( bool Enable )
{
    m_bAlpha = Enable;
}
```

CTextureFormatEnum::GetFormatSupported (Overloaded)

There may be times when you do not wish to use the default processing supplied by the GetBestFormat functions, but instead wish to query if a given format is supported. Your application can then use our object simply to test support for surfaces and make up its own mind about which one is preferred.

The GetFormatSupported function takes a single parameter of type D3DFORMAT that describes the surface you would like to query support for. The function then uses the STL map Find method to search the map for a matching key. Of course, the Enumerate method must have been called first for any of these query methods to work.

If the key exists in the map, then the iterator will be pointing at that key/value pair when the Find method returns. If not, the iterator will be pointing at the end of the map data and we know that we do not have a row in our map for the specified format. We have added all the common formats to the map, so this should never happen in the normal case (not unless you are using a custom format).

```
bool CTextureFormatEnum::GetFormatSupported( D3DFORMAT Format ) const
{
    FormatEnumMap::const_iterator Item = m_TextureFormats.find( Format );

    // We don't support the storage of this format.
    if ( Item == m_TextureFormats.end() ) return false;

    // Return the item
    return Item->second;
}
```

As you can see, we simply return the value of the iterator's 'second' member, which will be pointing to the boolean value for the key (the format) that was specified. Therefore, this function returns true or false to indicate support for that format.

CTextureFormatEnum::SetFormatSupported

The final function we will cover in this class is the SetFormatSupported function. The application can use this to force the boolean value associated with a format to be true or false. As the first parameter we pass the format (the key) whose value we wish to set. As the second parameter we pass a boolean that indicates the support (or lack of it) for the specified format.

```
void CTextureFormatEnum::SetFormatSupported( D3DFORMAT Format, bool Supported )
{
    FormatEnumMap::iterator Item = m_TextureFormats.find( Format );

    // We don't support the storage of this format.
    if ( Item == m_TextureFormats.end() ) return;

    // Update the item (done this way because we have already had to perform
    // a 'find'. It saves us a little time over using the indexing operator
    // to find the key again).
    Item->second = Supported;
}
```

We now have a texture format enumeration object that all our future components can use. All we have to do is make sure that any component we plug into our framework can be passed and store a pointer to the enumeration object. From that point on, the component can use this object to query for formats that it is interested in using.

Multiple Terrains in CScene

Since Chapter Three in Module I, our demonstration applications have generally taken one of two forms. They have either been terrain demos or non-terrain demos. The non-terrain demos used the CScene class as a geometry manager and loaded data from IWF files or X files. We have continued to use and expand the CScene class throughout Module II.

In previous terrain oriented demos, because the generation of the terrain geometry happened procedurally (based on heightmap data), a different scene manager was used, called CTerrain. The CTerrain class essentially replaced CScene in those demos. The data was no longer loaded from an IWF file and as such, lab projects that contained terrains used different code than those that did not. Although using the CTerrain class as a substitute scene manager for terrain based applications served us well in earlier lessons to keep the code simple and focused, this system will have to be revised going forward.

In a game we often do not want our scene to consist of either indoor or outdoor environments; often we want a scene to be comprised of both. Our current system does not support this idea, so we will upgrade the CScene and CTerrain classes so that they can be used together in a single application.

The CScene class will now officially be the top level scene manager and it will continue to load its data from IWF files. However, we will add support for the loading of terrain entities from the IWF file so that the IWF file can contain internal geometry and external references in addition to the terrain. In fact, a scene that we make in GILES™ may contain multiple terrain entities, so our scene will have to create, manage, and render multiple CTerrain objects.

> **Note**: You will see later that when a terrain exists as an entity in an IWF file, the geometry is not stored. All that is stored is the information used to construct it -- the heightmap file, the filenames of the base and detail textures, the scale of the detail texture and the scale of the geometry. These are all variables that were member variables in our old CTerrain class and were used to construct the terrain. Therefore, all we have to do is write a function that allows us to send these parameters into a terrain object when it is first created and we can use virtually the same code as before, with the exception of one or two little adjustments that will be discussed in a moment.

It may at first seem that changing the terrain class and the scene class to work harmoniously with one another would require many code changes. In the grand scheme of things, it requires hardly any. We will first discuss changes to the CScene class.

Just as the CScene class maintains an array of objects, meshes, and actors that collectively compose the virtual world, the CScene class will now also maintain an array of CTerrain objects. The following lines of code have been added to the class declaration of CScene:

Excerpt from CScene.h

```
CTerrain          **m_pTerrain;
ULONG               m_nTerrainCount;
```

As you can see, the scene now manages an array of CTerrain pointers and has an added ULONG describing how many CTerrain pointers are currently in the array. Every time the loading code encounters a terrain entity, it will create a new CTerrain object from the data extracted from the IWF file and add its pointer to this array.

CScene::AddTerrain

Just as the CScene class has an AddObject, an AddMesh, and an AddActor function to resize the respective arrays and make room for another element at the end, we now have to have an AddTerrain method which does the same for terrains.

This function is called by the IWF loading code whenever a terrain entity is encountered in the file. The code to this function should be easily understood since it is almost identical to all the other array resize functions we have written in previous lessons.

AddTerrain takes a single optional parameter that allows the caller to specify how many new elements should be added to the end when the array resize is performed. The default value of the parameter is 1, so this function will make space in the terrain array for one additional CTerrain pointer.

If previous terrain pointer data exists in the array it is copied over into the new resized array before the original array is deleted. The new resized array is then assigned to the CScene::m_pTerrain pointer replacing the original assignment to the old array.

```
long CScene::AddTerrain( ULONG Count /* = 1 */ )
{
    CTerrain ** pTerrainBuffer = NULL;

    // Allocate new resized array
    if (!( pTerrainBuffer = new CTerrain*[ m_nTerrainCount + Count ] )) return -1;

    // Existing Data?
    if ( m_pTerrain )
    {
        // Copy old data into new buffer
        memcpy( pTerrainBuffer, m_pTerrain, m_nTerrainCount * sizeof(CTerrain*) );

        // Release old buffer
```

```
        delete []m_pTerrain;

    } // End if

    // Store pointer for new buffer
    m_pTerrain = pTerrainBuffer;

    // Clear the new items
    ZeroMemory( &m_pTerrain[m_nTerrainCount], Count * sizeof(CTerrain*) );

    // Increase Terrain Count
    m_nTerrainCount += Count;

    // Return first Terrain
    return m_nTerrainCount - Count;
}
```

CScene::Render

As you are well aware, our scene is rendered when the CGameApp::FrameAdvance function calls the CScene::Render function. It is this function that instructs the various components comprising the scene to render themselves. It first instructs the skybox to render itself, and then it loops through every object in the scene instructing that object's mesh or actor to render itself.

We will now have to add an additional line to the CScene::Render function that instructs any terrains that may be used by the scene to render themselves as well. We will not cover the entire CScene::Render function at this time, as it has hardly changed at all. However, below we see a new line of code from that function which renders each terrain in the CScene::m_pTerrain array. This line of code is positioned very near the bottom of the function, just after the loop that renders each CObject.

```
// Render each terrain
for ( i = 0; i < m_nTerrainCount; ++i ) m_pTerrain[i]->Render( &Camera );
```

With the exception of the CScene IWF loading code upgrade which we will cover at the end of this workbook, these are the only changes that need to be added to the CScene class so that it now correctly renders multiple terrains. Next let us discuss the updates we will need to apply to our CTerrain class to get everything working together.

The New CTerrain Class

There is no need for concern since we do not have to write a new terrain class. With a few minor modifications, our old CTerrain class can be upgraded to work alongside our new CScene, and alongside other CTerrain objects that might exist in the scene.

> **Note**: This object was first covered in Chapter Three of Module I when we explained how the vertex data for the terrain was generated from a heightmap. We also described how the terrain was actually represented internally as a series of CMesh objects. A CMesh object was really just a lite wrapper around a vertex and index buffer with functions that allowed us to add vertices and indices incrementally to the object during the building process. As this object is no longer used anywhere else other than by the CTerrain class, we have decided to rename CMesh to CTerrainBlock. This is just a simple name change and all the code remains the same, unless stated otherwise. If you are feeling a little rusty on how we generated our terrain, please refer back to Chapter Three. You may also want to refresh yourself on the workbook of Module I Chapter Six where we discussed the concept of texturing the terrain with a base and detail texture map. All of this code will remain the same, so will not be covered again here.

The only real problem we need to address is that in our old CTerrain class the vertex data was assumed to be generated in world space. That is, the terrain space origin was also the world space origin. As the data was assumed to be in world space, rendering the terrain simply meant setting the world matrix to an identity matrix and sending the vertex and index data of each terrain block to the hardware.

In an IWF file, the terrain data is defined in model space and is accompanied by a world matrix describing where in the scene this terrain should be positioned when rendered or queried for collision. Therefore, we will have to add a world matrix member to our CTerrain class. Before the terrain renders itself (in the CTerrain::Render function) it will need to send this world matrix to the device. This simple addition means we can now load an IWF file that contains multiple terrain entities that are all placed at different locations in the world. We can thus position and render these terrains correctly.

Our CTerrain class will also have a method added called SetTextureFormat so that it can be passed (by CGameApp or CScene) a pointer to the CTextureFormatEnum object introduced earlier. The terrain creation functions can then use this object to find the optimal texture format to use when loading the base and detail texture map images.

Furthermore, we will upgrade our CTerrain object to provide lifetime encapsulation using COM-like semantics. That is, we will add the AddRef and Release mechanics so that the CTerrain object can destroy itself when its internal reference count reaches zero.

Finally, the LoadTerrain method will be written to replace the old LoadHeightMap method as the means for generating the terrain geometry. Previously, the application would call the CTerrain::LoadHeightMap method and pass the name of the .RAW file that contains the height data and the width and the height of that image. This function would then generate the terrain based on the heightmap data and the values of several internal variables (e.g., geometry scale, texture scale, etc.). If we wanted to change the generation properties of the terrain, we would have to alter the code and re-compile.

The new LoadTerrain method accepts a single parameter -- a pointer to a TerrainEntity structure. Not only does this single structure contain all the terrain creation parameters we used before, it also matches the format in which the data for a terrain is stored in an IWF file. Therefore, when our loading code encounters a terrain entity, it can just pull the data from the file into a TerrainEntity structure and pass it straight to the CTerrain::LoadTerrain method for geometry generation. This is very handy indeed.

Below we show the new methods added to CTerrain in the file 'CTerrain.h'. Not all of the class methods are show here in order to keep the listing compact, but we have included the ones that are new and those that have been modified. We have also shown the member variables and highlighted new or modified ones in bold face type.

```
class CTerrain
{
public:
    //-----------------------------------------------------------------------
    // Constructors & Destructors for This Class
    //-----------------------------------------------------------------------
    CTerrain();
    virtual ~CTerrain();


    // Public Functions For This Class
    …
    …
    …
    void                SetTextureFormat ( const CTextureFormatEnum &FormatEnum );
    void                SetWorldMatrix   ( const D3DXMATRIX & mtxWorld );

    bool                LoadTerrain      ( TerrainEntity * pTerrain );
    const D3DXMATRIX &  GetWorldMatrix   ( ) const {return m_mtxWorld; }
    …
    …
    …

    // Reference Counting Functions
    ULONG               AddRef              ( );
    ULONG               Release             ( );


    // Public Static Functions For This Class (callbacks)
    static void UpdatePlayer(LPVOID pContext, CPlayer *pPlayer, float TimeScale );
    static void UpdateCamera(LPVOID pContext, CCamera *pCamera, float TimeScale );

private:

    // Private Variables For This Class

    D3DXVECTOR3         m_vecScale;         // Amount to scale the terrain meshes
    D3DXVECTOR2         m_vecDetailScale;   // Amount to scale the detail map.
    float              *m_pHeightMap;       // The physical heightmap data loaded
    ULONG               m_nHeightMapWidth;  // Width of the 2D heightmap data
    ULONG               m_nHeightMapHeight; // Height of the 2D heightmap data

    CTerrainBlock     **m_pTerrainBlocks;   // Simple array of mesh pointers
```

```
    ULONG               m_nBlockCount;      // Number of blocks stored here
    LPDIRECT3DDEVICE9   m_pD3DDevice;       // D3D Device
    bool                m_bHardwareTnL;     // Used hardware vertex processing ?
    bool                m_bSinglePass;      // Use single pass rendering method

    CTextureFormatEnum  m_TextureFormats;   // Texture format lookup object

    LPTSTR              m_strDataPath;      // Path to data items
    ULONG               m_nPrimitiveCount;  // Number of primitives for D3D Render
    LPDIRECT3DTEXTURE9  m_pBaseTexture;     // Base terrain texture
    LPDIRECT3DTEXTURE9  m_pDetailTexture;   // Terrain detail texture.

    D3DXMATRIX          m_mtxWorld;         // Terrain world matrix
    ULONG               m_nRefCount;        // Reference Count Variable

    // Private Functions For This Class
    long                AddTerrainBlock     ( ULONG Count = 1 );
    bool                BuildTerrainBlocks  ( );
    void                FilterHeightMap     ( );
};
```

There are a couple things to pay attention to in the above listing. The terrain now has a world matrix and also has Get/Set functions that allow for the configuration of this matrix by external components. It also has a reference count variable with accompanying AddRef/Release methods. So the terrain now behaves like a COM object and destroys itself when there are no outstanding references to it. Finally, notice that what was once a CMesh pointer array containing the terrain blocks is now an array of CTerrainBlocks. This is just an object renaming for clarity and no code has changed. Thus the function AddMesh has had its name changed to AddTerrainBlock and the function BuildMeshes has had its name changed to BuildTerrainBlocks.

The TerrainEntity Structure

The TerrainEntity structure is defined in CTerrain.h and is our property transport structure when informing the CTerrain object how its geometry should be built. It also describes exactly how the terrain data exported from GILES™ into the IWF file is laid out in that file.

We will see later when we cover the upgraded loading code that for each terrain entity found in the file we will create a new CTerrain object and pass a structure of this type to its LoadTerrain function. Of course, you can fill one of these structures out manually if you wish to generate your terrain creation properties from another source and not use GILES™ or IWF files. The members of this structure should be instantly familiar to you since they are the properties that existed in our previous CTerrain class member variables. We will briefly discuss the members as a reminder.

```
typedef struct _TerrainEntity
{
    ULONG       TerrainType;        // Which type of terrain are we using.
    char        TerrainDef[256];    // Either terrain definition (splats) or
                                       heightmap def (standard).
```

```
    char        TerrainTex[256];    // (Standard terrain only), specifies the base
                                        texture
    char        TerrainDet[256];    // (Standard terrain only), specifies the
                                        detail texture
    ULONG       TerrainWidth;       // (Standard terrain only), Width of the
                                        heightmap
    ULONG       TerrainHeight;      // (Standard terrain only), Height of the
                                        heightmap
    D3DXVECTOR3 TerrainScale;       // (Standard terrain only), Scale of the
                                        terrain
    D3DXVECTOR2 TerrainDetScale;    // (Standard terrain only),
                                    //  scale information for detail texture
    ULONG       Flags;              // Reserved Flags

} TerrainEntity;
```

ULONG TerrainType
The GILES™ terrain entity can be one of two types -- a regular terrain or a splatting terrain. You will recall that the data is generated rather differently for each terrain type. Our CTerrain object encapsulates regular heightmapped terrains. This value will be stored in the entity to inform the loading code which type of terrain entity has been encountered. A value of zero indicates a regular terrain entity. Therefore, our CScene class will only be interested in loading regular terrain entities, as that is the type of terrain this object currently encapsulates. Thus, this value should always be zero when this structure is used to create CTerrain objects (at least for now).

char TerrainDef[256]
This member contains the filename of the heightmap image that the CTerrain object will need to load and extract the height values from. For splatting terrains, this will contain the name of the splat definition file that contains all the splat terrain info.
(The following members are applicable only to regular terrains)

char TerrainTex[256]
This member contains the filename of the image file that should be loaded into a texture and used as the base texture for the terrain.

char TerrainDet[256]
This member contains the filename of the image file that should be loaded into a texture and used to tile over the terrain as a detail texture.

ULONG Width
ULONG Height
As the image data for the terrain heightmap is stored in a .RAW file that contains no data describing the arrangements of the pixels in the image, these members inform us how the data in the .RAW file is to be interpreted in terms of rows and columns. For example, if Width=1024 and Height=512, the pixels in the .RAW file should be used to create a mesh of 512 rows of 1024 vertices.

D3DXVECTOR3 TerrainScale
This vector describes how the pixel positions in the heightmap are scaled into terrain space vertex positions. For example, a scale vector of (1,1,1) would produce a terrain where every unit in world space maps to every pixel in image space and the height data for any pixel will be used exactly as the world space Y component of that vertex. A scale vector of (1,5,2) on the other hand, would scale the pixel position by 1 along the X axis, 2 along the Z axis, and the Y component of the vertex would be generated by scaling the value stored in the pixel by 5. Using this second example scale vector, a raw file containing 10 rows and 10 columns would create a terrain 10 units wide (X axis) and 20 units deep (Z axis) and at its peak would be five times higher than the highest point described by the height data.

D3DXVECTOR2 TerrainDetScale
This 2D vector describes how many times we would like the detail texture tiled over the terrain in the direction of the X and Z axes.

CTerrain::LoadTerrain

The LoadTerrain method is used to instruct a newly created CTerrain object to build its geometry and load its texture resources. This function is almost identical to the LoadHeightMap method in our old CTerrain class with the exception that it now extracts the properties for creation from the passed TerrainEntity structure.

We will show the code to this function below but provide only very brief descriptions since it should be familiar to you. We will not show the code to any of the helper functions it calls since these were all covered in Module I and are unchanged. Please refer back to Chapters Three and Six if you need to jog your memory about how these functions work. This function is shown as a reminder of the terrain creation process from a high level.

The first thing this function does is return if the TerrainType member of the passed TerrainEntity structure is not zero. This means it is not a regular terrain and as such, not a terrain this class was designed to support. We also return if the device is not valid or if its array of terrain blocks has already been allocated. If data already exists in the terrain, it should be released prior to calling this function. We then extract the width and height values from the passed structure and store them in member variables.

```
bool CTerrain::LoadTerrain( TerrainEntity * pTerrain )
{
    HRESULT   hRet;
    FILE    * pFile = NULL;
    TCHAR     FileName[MAX_PATH];
    D3DFORMAT fmtTexture;
    ULONG     i;

    // Skip if this is not a 'standard' terrain
    if ( !pTerrain || pTerrain->TerrainType != 0 ) return false;

    // Cannot load if already allocated (must be explicitly released for reuse)
    if ( m_pTerrainBlocks ) return false;

    // Must have an already set D3D Device
```

```
    if ( !m_pD3DDevice ) return false;

    // First of all store the information passed
    m_nHeightMapWidth  = pTerrain->TerrainWidth;
    m_nHeightMapHeight = pTerrain->TerrainHeight;
```

In the next section we combine the name of the highway file with the string containing our application's data path so that we have the complete path of the RAW file we wish to load. We then open the file and read in its total size, so that we know how much data we will need to read.

```
    // Build full filename
    _tcscpy( FileName, (m_strDataPath) ? m_strDataPath : pTerrain->TerrainDef );
    if ( m_strDataPath ) _tcscat( FileName, pTerrain->TerrainDef );

    // Open up the heightmap file
    pFile = _tfopen( FileName, _T("rb") );
    if (!pFile) return false;

    // Get file length
    ULONG FileSize = filelength( pFile->_file );
```

We next calculate how many rows and columns our heightmap must have using the passed width and height values in the TerrainEntity structure. For example, if FileSize contains the number of pixels in the image, dividing this by the height of the heightmap will tell us the length of a single row. The number of rows is then calculated by dividing the file size by the length of a row. We also perform some logic to calculate the height and width if only one of them has been specified in the terrain entity structure (and the other property is set to zero). If the TerrainEntity structure has zero in both its width and height members, then we will assume the image file is to be interpreted as a square heightmap. In that case the width and height will be equal to the square root of the total file size.

```
    // Work out heightmap size if possible
    if ( m_nHeightMapWidth > 0 && m_nHeightMapHeight == 0 )
    {
        // We can work this one out
        m_nHeightMapHeight = FileSize / m_nHeightMapWidth;

    } // End if m_nHeightMapWidth only
    else if ( m_nHeightMapWidth == 0 && m_nHeightMapHeight > 0 )
    {
        // We can work this one out
        m_nHeightMapWidth = FileSize / m_nHeightMapHeight;

    } // End if m_nHeightMapHeight only
    else
    {
        // We can only assume square
        m_nHeightMapWidth  = (ULONG)sqrt( (double)FileSize );
        m_nHeightMapHeight = m_nHeightMapWidth;

    } // End if no sizes at all
```

In the next step we test that the file size is large enough to have enough data for the width and height values we have just calculated. If the file size is smaller than 'Width x Height' then something has gone wrong and we return. We also retrieve the terrain geometry scale and the detail texture scale vectors and copy them into member variables.

```
    // Validate sizes
    if ( m_nHeightMapWidth * m_nHeightMapHeight > FileSize )
        { fclose( pFile); return false; }

    // Retrieve scale
    m_vecScale = pTerrain->TerrainScale;
    m_vecDetailScale = pTerrain->TerrainDetScale;
```

Now we allocate an array of floats large enough to store a floating point value for each pixel in the heightmap. This is the array we will copy the height values into for each vertex of the terrain we are about to create. We then set up a loop to read the value of each pixel and store it in the heightmap array before we close the file.

```
    // Attempt to allocate space for this heightmap information
    m_pHeightMap = new float[m_nHeightMapWidth * m_nHeightMapHeight];
    if (!m_pHeightMap) return false;

    // Read the heightmap data
    for ( i = 0; i < m_nHeightMapWidth * m_nHeightMapHeight; i++ )
    {
        UCHAR HeightValue;
        fread( &HeightValue, 1, 1, pFile );

        // Store it as floating point
        m_pHeightMap[i] = (float)HeightValue;

    } // Next Value

    // Finish up
    fclose( pFile );
```

We now call the FilterHeightMap method which applies a smoothing filter to the height data so that our terrain does not look jagged because of the integer height values stored in the file. That is why we read the integer height data into a float array -- once we have the height data stored as floats we can apply a smoothing filter to soften the peaks and valleys. The code to this function was covered in Module I and is unchanged.

```
    // Filter the heightmap data
    FilterHeightMap();
```

Our next step is to load the base texture and the detail texture. Before calling the D3DXCreateTextureFromFileEx function to load these textures, we call the GetBestFormat method of our CTextureFormatEnum object. This will return a D3DFORMAT type that we can pass to the loading functions for texture generation. Before we load each texture, we append the filename to the

application's data path so we have the full path and filename of the image file to pass to the texture loading function.

```
    // Get the best format for these textures
    fmtTexture = m_TextureFormats.GetBestFormat( );

    // Load in the textures used for rendering the terrain
    _tcscpy( FileName, (m_strDataPath) ? m_strDataPath : pTerrain->TerrainTex );
    if ( m_strDataPath ) _tcscat( FileName, pTerrain->TerrainTex );
    hRet = D3DXCreateTextureFromFileEx( m_pD3DDevice,
                                        FileName,
                                        D3DX_DEFAULT,
                                        D3DX_DEFAULT,
                                        D3DX_DEFAULT,
                                        0,
                                        fmtTexture,
                                        D3DPOOL_MANAGED,
                                        D3DX_DEFAULT,
                                        D3DX_DEFAULT,
                                        0, NULL, NULL, &m_pBaseTexture );


    _tcscpy( FileName, (m_strDataPath) ? m_strDataPath : pTerrain->TerrainDet );
    if ( m_strDataPath ) _tcscat( FileName, pTerrain->TerrainDet );
    hRet = D3DXCreateTextureFromFileEx(   m_pD3DDevice,
                                          FileName,
                                          D3DX_DEFAULT,
                                          D3DX_DEFAULT,
                                          D3DX_DEFAULT,
                                          0,
                                          fmtTexture,
                                          D3DPOOL_MANAGED,
                                          D3DX_DEFAULT, D3DX_DEFAULT,
                                          0,
                                          NULL,
                                          NULL,
                                          &m_pDetailTexture );
```

At this point the heightmap data is loaded and filtered and the texture assets have been loaded. All that is left to do is generate the terrain blocks.

We first call the AddTerrainBlock method to allocate enough space in the terrain block pointer array for each terrain block we will create. The number of terrain blocks the heightmap data is divided into depends on how many quads we wish to place in each block. The QuadsWide and QuadsHigh variables can be altered and are also defined in CTerrain.h. By default, they are set to 16 so that each terrain block contains a 16x16 block of quads.

```
    // Allocate enough terrain blocks to store the separate blocks of this terrain
    if ( AddTerrainBlock( ((m_nHeightMapWidth - 1) / QuadsWide) *
                          ((m_nHeightMapHeight - 1)/ QuadsHigh) ) < 0 )
       return false;
```

The AddTerrainBlock function is not a new function. It was previously called AddMesh but has now been renamed for consistency with our naming scheme.

Finally, we call the BuildTerrainBlocks function, which is just a renamed version of the old BuildMeshes method. It is this function that generates the geometry for each terrain block.

```
    // Build the terrain block data itself
    return BuildTerrainBlocks( );
}
```

CTerrain::Render

The Render method in the CTerrain class has an additional line now. You will remember that the render function essentially just binds the base and detail textures to stages 0 and 1 on the device and then iterates through each terrain block setting its vertex and index buffers on the device and then rendering them. The main difference now is that we must set the terrain's world matrix prior to rendering it since the vertices are now defined in terrain (model) space. The following line has been inserted very near the top of the CTerrain::Render function before we loop through and render each block.

```
// Set this terrain's transformation
m_pD3DDevice->SetTransform( D3DTS_WORLD, &m_mtxWorld );
```

Note: The CTerrain class exposes methods that allow the application to set and retrieve the world matrix.

Before each terrain block is rendered we also perform a frustum test on it, just as we did before. When we call the CCamera::BoundsInFrustum function to perform this test, we also pass the world matrix along with the bounding box min and max vectors. Since the terrain block's bounding box is in terrain space it must be converted into world space for the test. The BoundsInFrustum method of CCamera has an optional third matrix parameter that, when used, will cause the function to transform the bounding box extents using the passed world matrix prior to performing the frustum test.

The line of code that is executed before we process each block in the CTerrain::Render function is shown below. It is the first line that is executed in the loop that processes each block. If the terrain block is not visible, the current iteration of the loop is skipped.

```
if ( pCamera && (!pCamera->BoundsInFrustum( m_pTerrainBlocks[i]->m_BoundsMin,
                                            m_pTerrainBlocks[i]->m_BoundsMax,
                                            &m_mtxWorld )) ) continue;
```

Other Changes to CTerrain

Before leaving this section, we will point out other small changes that have taken place to CTerrain. Most are so small, that to cover the function code again would be redundant.

You will recall from Module I that our CTerrain class registered two static callback functions with the CPlayer object that were used for collision detection purposes. These functions were called UpdatePlayer and UpdateCamera. They were called by CPlayer whenever it wanted to update the position of the player or its attached camera. If the position of either was found to be below the terrain, it would be adjusted so that it lay on the terrain surface (see Chapter Three).

The function involved finding the quad over which the player was positioned and interpolating the height values to find the exact height of the terrain at the new position. If this was higher than the player's position, then the position of the player (or its camera) was updated.

These functions must be updated slightly because we can no longer compare the position of the player against the interpolated vertex heights as they no longer exist in the same space; the player/camera position is in world space and the vertex data of the terrain is in terrain space. In order for these functions to continue working correctly we must add an additional line to the top of each function that transforms the passed player/camera position into terrain space. This is done by multiplying the player position with the inverse world matrix of the terrain. The function will then continue on as usual.

Updating the CScene IWF Loading Code

In this lab project, the IWF loading code in CScene will be upgraded to support two additional entity types: Terrain and Fog. Since these are both entities, this will amount to nothing more than adding two addition case statements to the CScene::ProcessEntities function.

We will show the ProcessEntities function in a moment. Note that we will snip out the code that processes references we are familiar with (light, external reference, and skybox) although we will leave the case statements in place so you can see the general flow of the function.

CScene and DirectX Fog

Although we have been able to add fog to our scenes since Chapter Seven of Module I, until this lab project, we were forced to hardcode the fog parameters and generate the results we wanted through tedious trial and error. GILES™ allows us to configure fog and view it in an easy to tweak manner, so it is a good tool for experimenting with our fog parameters (without having to recompile code).

GILES™ saves the fog settings for the scene in an entity chunk with the entity ID seen below. This is accompanied by the GILES™ author ID. We will define this value in CScene.h so that we can easily test for this entity type when searching through the entity vector.

```
#define CUSTOM_ENTITY_FOG       0x204
```

The data area for a fog entity is laid out as shown in the following structure. This structure is also defined in CScene.h and will be used to contain the fog data we fetch from the IWF file.

Excerpt from CScene.h

```
typedef struct _FogEntity
{
    ULONG       Flags;                  // Fog type flags
    ULONG       FogColor;               // Color of fog.
    ULONG       FogTableMode;           // Mode to use with table fog
    ULONG       FogVertexMode;          // Mode to use with vertex fog
    float       FogStart;               // Fog range start
    float       FogEnd;                 // Fog range end
    float       FogDensity;             // Density float

} FogEntity;
```

We know from our coverage of DirectX fog in Chapter Seven what most of these values mean and how they contribute to the fog factor, but there are a few members here that need explanation since their meaning is specific to this type of entity.

ULONG Flags
These flags can be zero or one of the following modifier flags that help inform us about what type of fog is preferred by the IWF file creator. There are two flags defined in CScene.h that map to these values.

```
#define FOG_USERANGEFOG         0x1
```

If this flag is set it means that if our scene is going to use vertex fog instead of table fog then range based fog is preferred (as is normally the case). This flag is only valid when vertex fog is used as there is no range based pixel fog in DirectX 9.

```
#define FOG_PREFERTABLEFOG      0x2   // Modifier used for FogEntity::Flags member
```

When the flags member of the entity has this flag set, it means the IWF creator would prefer to use table (pixel) fog. This is generally the preferred mode for most applications as it gives the best results (see Chapter Seven of Module I).

ULONG FogTableMode
ULONG FogVertexMode
These two members of the entity data area define the fog models the artist would like us to use for vertex and table fog modes. They can be set to any of the values you see below. You should understand what these definitions mean and the way the fog factor is computed in each of the fog models since we covered them in Chapter Seven. These values are also defined in CScene.h

```
#define FOGMODE_NONE            0x0         // Mode flags for FogEntity
#define FOGMODE_EXP             0x1         // ''
#define FOGMODE_EXP2            0x2         // ''
#define FOGMODE_LINEAR          0x3         // ''
```

Remember, both the FogTableMode and the FogVertexMode will be set to one of these values. Although an application will typically use only one of these fog modes, specifying the options for each allows us more control over specifying what should be used when one is not supported.

The CScene class itself actually has a 'FogEntity' structure as a member. It is this variable that will be populated with the fog entity data extracted from the IWF file. Below we see the two new members that have been added to CScene to contain any fog data that has been loaded.

```
bool                m_bFogEnabled;    // Is fog enabled or not in this level?
FogEntity           m_FogEntity;      // Storage for our fog setup.
```

When the m_FogEntity structure is populated with fog data from the IWF file, the m_bFogEnabled boolean is also set to true; otherwise it stays set as false. Forgetting about the actual loading code for just a moment, the CScene::Render function has the following lines inserted near the start of the function to set up the fog parameters prior to rendering any objects:

Excerpt from CScene::Render

```
if ( m_bFogEnabled )
    {
        // Setup fog parameters
        m_pD3DDevice->SetRenderState( D3DRS_FOGCOLOR  , m_FogEntity.FogColor);
        m_pD3DDevice->SetRenderState( D3DRS_FOGSTART  , *(ULONG*)&m_FogEntity.FogStart  );
        m_pD3DDevice->SetRenderState( D3DRS_FOGEND    , *(ULONG*)&m_FogEntity.FogEnd    );
        m_pD3DDevice->SetRenderState( D3DRS_FOGDENSITY, *(ULONG*)&m_FogEntity.FogDensity);
        m_pD3DDevice->SetRenderState( D3DRS_RANGEFOGENABLE,
                                     (m_FogEntity.Flags & FOG_USERANGEFOG)? TRUE : FALSE );
        m_pD3DDevice->SetRenderState( D3DRS_FOGENABLE, TRUE );
```

The fog color, start and end distances, and density are taken straight from the CScene::m_pFogEntity structure and passed to the pipeline. We also enable the device's fog render state. Notice that we only enable range based fog if the FOG_USERANGEFOG flag has been set in the entity structure.

The final section of the fog render code is shown below. If table fog is preferred then we enable it and disable the vertex fog mode (and vice versa).

Excerpt from CScene::Render Continued

```
        // Set up conditional table / vertex modes
        if ( m_FogEntity.Flags & FOG_PREFERTABLEFOG )
        {
            m_pD3DDevice->SetRenderState( D3DRS_FOGTABLEMODE, m_FogEntity.FogTableMode );
            m_pD3DDevice->SetRenderState( D3DRS_FOGVERTEXMODE, D3DFOG_NONE );

        } // End if table fog
        else
        {
            m_pD3DDevice->SetRenderState( D3DRS_FOGVERTEXMODE, m_FogEntity.FogVertexMode );
            m_pD3DDevice->SetRenderState( D3DRS_FOGTABLEMODE, D3DFOG_NONE );

        } // End if vertex fog

    } // End if fog enabled
```

Looking at the above two sections of code it may at first seem as if we are not validating the requested fog techniques on the device to see if they are supported. It looks like we are essentially just extracting the information in the FogEntity structure that was loaded from the file and trying to use them. However, you will see in a moment when we look at the code that loads the fog entity that, after the data

has been loaded into the FogEntity structure, the device is tested and the members are modified to reflect what can and cannot be used on the current device.

CScene::ProcessEntities (Updated)

This function has been in use since Module I (to parse light entities). Along the way we added support for external reference entities, sky boxes, and most recently, trees. Because these sections of the function have already been covered, the code that parses these entities is not shown again here to compact the listing. In such places where code has been removed we have replaced the code with the line '……snip……'.

As we know, this function is called from CScene::LoadSceneFromIWF after the data from the IWF file has been loaded into the CFileIWF object's internal STL vectors. This function's job is to extract the entity information from that object into a format that can be used by the application. The function sets up a loop to visit each entity in the CFileIWF object's entity vector. Each entity in this vector is of type iwfEntity, which is essentially a structure that contains a world matrix, the entity name, and a data area for storing arbitrary data. It also stores an entity type ID member and an author ID.

```
bool CScene::ProcessEntities( const CFileIWF& File )
{
    ULONG      i, j;
    D3DLIGHT9  Light;
    USHORT     StringLength;
    bool       SkyBoxBuilt = false;
    bool       FogBuilt    = false;

    // Loop through and build our lights & references
    for ( i = 0; i < File.m_vpEntityList.size(); i++ )
    {
        // Retrieve pointer to file entity
        iwfEntity * pFileEntity = File.m_vpEntityList[i];

        // Skip if there is no data
        if ( pFileEntity->DataSize == 0 ) continue;
```

The first thing we do inside the entity loop is get a pointer to the current entity we wish to process. We also test the DataSize member and if it is zero, then the data area is empty and we skip it.

The first entity we test for is the light entity, which is part of the IWF SDK specification. The standard light entity has an entity ID of 0x0010, so if we find an entity with this ID we know we have found a standard light. In the following code you can see that we only continue to parse the light entity if we have not already set up the maximum number of lights we wish to use. As this is not new code to us, the code that extracts the light information from the entity is not shown here.

```
        if ( (m_nLightCount < m_nLightLimit && m_nLightCount < MAX_LIGHTS)
              && pFileEntity->EntityTypeMatches( ENTITY_LIGHT ) )
        {
            //..... snip .....
        } // End if light
```

The light entity is the only entity type we currently process which is part of the core IWF standard. The rest of the entities we will load will be custom entities with the 'GILES' author ID. We can use the iwfEntity object's EntityAuthorMatches method and pass it a char array containing the ID we are looking for. In the following code, AuthorID has been defined at the start of 'CScene.cpp' as a 5 element char array containing the letters 'G','I','L','E','S'. This is the author ID GILES™ assigns to its custom entity chunks. If this function returns true, then the current entity is a GILES™ entity and we will initiate some logic to parse this entity.

```
        else

        if ( pFileEntity->EntityAuthorMatches( 5, AuthorID ) )
        {
            CTerrain * pNewTerrain = NULL;

            SkyBoxEntity SkyBox;
            ZeroMemory( &SkyBox, sizeof(SkyBoxEntity) );

            ReferenceEntity Reference;
            ZeroMemory( &Reference, sizeof(ReferenceEntity) );

            TerrainEntity Terrain;
            ZeroMemory( &Terrain, sizeof(TerrainEntity) );

            FogEntity Fog;
            ZeroMemory( &Fog, sizeof(FogEntity) );
```

Notice in the above code that we instantiate a temporary TerrainEntity structure and a FogEntity structure in addition to the others we previously supported. These structures are used to hold the data we extract.

Next we retrieve a pointer to the entity data area before entering a switch statement that will determine which of the GILES™ custom entities we are dealing with here. Since we have covered the code that processes the reference entity and the sky box entity in previous lessons, the code has been snipped to increase readability.

```
            // Retrieve data area
            UCHAR * pEntityData = pFileEntity->DataArea;

            switch ( pFileEntity->EntityTypeID )
            {
                case CUSTOM_ENTITY_REFERENCE:

                    // ..... snip .....

                    break;

                case CUSTOM_ENTITY_SKYBOX:

                    // ..... snip .....

                    break;
```

Now we add some new content. In the next section of code we add a case statement to test if the current entity has an ID of CUSTOM_ENTITY_TERRAIN. This value is defined in CTerrain.h as:

```
#define CUSTOM_ENTITY_TERRAIN   0x201
```

The value 0x201 is the ID that GILES™ assigns its terrain entity so we know that if this case is executed, it is a GILES™ terrain entity we are processing. Here is the first section of the code that processes the terrain entity.

```
                case CUSTOM_ENTITY_TERRAIN:

                    // Copy over the terrain data
                    memcpy( &Terrain.TerrainType, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);
```

Earlier in the function we fetched a pointer to the beginning of the entity data area. This data area is laid out in an identical manner to our TerrainEntity structure, so all we have to do is extract the values from the data area into one of these structures. In the above code we copy the first four bytes of the data area which contains the TerrainType value into the TerrainType member of our TerrainEntity structure. We then advance the data pointer past the value we have just extracted so that it is pointing at the beginning of the second piece of data to be extracted.

Before continuing to extract the rest of the data we should test the terrain type value we just extracted. Our CTerrain class encapsulates standard terrains (type 0) so we are currently only interested in parsing those.

In the following code we show a similar technique for copying the data to that which was described above. Each time, we copy the data into the TerrainEntity structure and advance the pointer by the correct number of bytes to point at the next piece of data. The following lines of code show us copying over the terrain Flags, its Width and its Height into their respective members in the TerrainEntity structure.

```
                    // Terrain, per-type processing
                    if ( Terrain.TerrainType == 0 )
                    {
                        memcpy( &Terrain.Flags, pEntityData, sizeof(ULONG) );
                        pEntityData += sizeof(ULONG);

                        memcpy( &Terrain.TerrainWidth,
                                pEntityData, sizeof(ULONG) );

                        pEntityData += sizeof(ULONG);

                        memcpy( &Terrain.TerrainHeight,
                                pEntityData, sizeof(ULONG) );

                        pEntityData += sizeof(ULONG);
```

After the width and height values in the file there is an unsigned short value informing us of the length of the filename string that follows (the heightmap image file). First we copy this value into a temporary

value called StringLength so we know how long the filename that follows will be, then we advance the data pointer to point at the beginning of this string, and finally we copy the filename from the data area into the TerrainDef character array of the TerrainEntity structure. This is shown below.

```
memcpy( &StringLength, pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

if ( StringLength > 0 )
   memcpy(Terrain.TerrainDef, pEntityData, StringLength );

pEntityData += StringLength;
```

All strings stored in the entity are defined in this way (i.e., prior to the string data will be an integer value describing the length of the string that follows).

After the filename of the heightmap in the data area, we will find two more strings (with their associated lengths) containing the filenames of the base and detail textures. Once again, the same technique is used for copying into the TerrainTex and TerrainDet members of our TerrainEntity structure.

```
memcpy( &StringLength, pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

if ( StringLength > 0 )
memcpy( Terrain.TerrainTex, pEntityData, StringLength );
pEntityData += StringLength;

memcpy( &StringLength, pEntityData, sizeof(USHORT) );
pEntityData += sizeof(USHORT);

if ( StringLength > 0 )
memcpy( Terrain.TerrainDet, pEntityData, StringLength );
pEntityData += StringLength;
```

The final two pieces of data to extract from the data area are the terrain scale (3D vector) and the detail map scale (2D vector). The following code copies these into the TerrainScale and TerrainDetScale members of the TerrainEntity structure.

```
memcpy( &Terrain.TerrainScale, pEntityData,
         sizeof(D3DXVECTOR3) );
pEntityData += sizeof(D3DXVECTOR3);

memcpy( &Terrain.TerrainDetScale, pEntityData,
        sizeof(D3DXVECTOR2) );
pEntityData += sizeof(D3DXVECTOR2);
```

At this point we have extracted all the data from the terrain entity data area and now have it stored in a TerrainEntity structure. We will now generate the terrain object.

First we call the CScene::AddTerrain method to make space at the end of the scene's CTerrain pointer array for an additional pointer. We then allocate a new CTerrain object.

```
                        // Add a new slot ready for the terrain
                        if ( AddTerrain( 1 ) < 0 ) break;

                        // Allocate a new terrain object
                        pNewTerrain = new CTerrain;
                        if ( !pNewTerrain ) break;
```

We send the terrain object the 3D device and the CTextureFormatsEnum object we are using so it can store the pointers internally for its own use. We also call the SetWorldMatrix method and send it the entity's matrix. We then set the data path so that the terrain knows where to find its texture images and heightmap files. Finally, we store the pointer of our new CTerrain object at the end of the scene object's terrain array.

```
                        // Setup the terrain
                        pNewTerrain->SetD3DDevice( m_pD3DDevice, m_bHardwareTnL );
                        pNewTerrain->SetTextureFormat( m_TextureFormats );
                        pNewTerrain->SetRenderMode(GetGameApp()->GetSinglePass());
                        pNewTerrain->SetWorldMatrix((D3DXMATRIX&)
                                                    pFileEntity->ObjectMatrix );

                        pNewTerrain->SetDataPath( m_strDataPath );

                        // Store it
                        m_pTerrain[ m_nTerrainCount - 1 ] = pNewTerrain;
```

We now call the CTerrain::LoadTerrain method passing in the TerrainEntity structure that we have filled out. We know that this is the function that will generate the terrain geometry. We also register the CTerrain player and camera callbacks with the player object (used for collision purposes) and our job is done.

```
                        // Load the terrain
                        if ( !pNewTerrain->LoadTerrain( &Terrain ) ) return false;

                        // Add the callbacks for collision with the terrain
                        GetGameApp()->GetPlayer()->AddPlayerCallback(
                                     CTerrain::UpdatePlayer, pNewTerrain );

                        GetGameApp()->GetPlayer()->AddCameraCallback(
                                           CTerrain::UpdateCamera, pNewTerrain );

                    } // End if 'standard' terrain

                    break;
```

The next and final section of this function is executed if the entity being processed is a GILES™ fog entity. The GILES™ fog entity has an entity ID of 0x204 and we set up a define in CScene.h as shown below:

```
#define CUSTOM_ENTITY_FOG       0x204
```

The first thing we do inside the fog case block is test a boolean variable called FogBuilt. If it was already set to true we skip the entity because it is possible that multiple fog entities might exist in the IWF file and we are only interested in finding and using the first one. Once we have processed this fog entity, the boolean will be set to true and any other fog entities will be ignored.

```
                case CUSTOM_ENTITY_FOG:

                    // We only want one fog setup per file please! :)
                    if ( FogBuilt == true ) break;
                    FogBuilt = true;
```

The data area of the fog entity is laid out in exactly the same way as the FogEntity structure we discussed earlier, so we will extract the data straight into this structure. Here is the remaining code to the function.

```
                    // Retrieve the fog entity details.
                    memcpy( &Fog.Flags, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Fog.FogColor, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Fog.FogTableMode, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Fog.FogVertexMode, pEntityData, sizeof(ULONG) );
                    pEntityData += sizeof(ULONG);

                    memcpy( &Fog.FogStart, pEntityData, sizeof(float) );
                    pEntityData += sizeof(float);

                    memcpy( &Fog.FogEnd, pEntityData, sizeof(float) );
                    pEntityData += sizeof(float);

                    memcpy( &Fog.FogDensity, pEntityData, sizeof(float) );
                    pEntityData += sizeof(float);

                    // Process the fog entity (it requires validation etc)
                    if ( !ProcessFog( Fog ) ) return false;

                    break;

                } // End Entity Type Switch

            } // End if custom entities

        } // Next Entity

        // Success!
        return true;
}
```

Notice that after we have extracted the fog information into the FogEntity structure, we call a new member function of CScene called ProcessFog.

CScene::ProcessFog

This function is passed the FogEntity structure that contains all the fog settings loaded from the fog entity. This function has the task of testing the requested fog settings to see if they are supported by the device and modifying any settings that are not.

```
bool CScene::ProcessFog( const FogEntity & Fog )
{
    D3DCAPS9  Caps;
    CGameApp *pApp          = GetGameApp();
    bool      bTableSupport = false, bVertexSupport = false;
    bool      bRangeSupport = false, bWFogSupport   = false;

    // Disable fog here just in case
    m_bFogEnabled = false;

    // Validate requirements
    if ( !m_pD3DDevice || !pApp ) return false;

    // Store fog entity
    m_FogEntity = Fog;

    // Retrieve the card capabilities (on failure, just silently return)
    if ( FAILED( m_pD3DDevice->GetDeviceCaps( &Caps ) ) ) return true;
```

The first section of the function shown above starts by setting CScene::m_bFogEnabled to false. We do this just in case fog is not supported. If this is not set to true at some point in the course of this function the render function for CScene will not set any fog parameters. We also store the passed FogEntity structure in the CScene member variable m_FogEntity since this is the structure the CScene::Render function will use to set up the fog parameters for the DirectX pipeline. Then we call the IDirect3DDevice9::GetDeviceCaps method to get the capabilities of the device.

In the next section of code we query four device capabilities and set four local booleans to true or false based on whether they are supported are not. We need to test support for four fog settings:

1. Is table/pixel fog supported on this device?
2. Is vertex fog supported on this device?
3. If vertex fog is supported does it support the more desirable range based vertex fog (instead of view space Z depth fog)?
4. If table fog is supported does it support the use of the W component in its fog calculations (view space Z) or does it use 0.0-1.0 device coordinates (Z buffer coordinates). W based table fog is more desirable for two reasons. First, the Z buffer coordinates have a very non-linear distribution of depth values causing flaky fog effects for objects in the far distance. Second, it is much nicer when using the linear fog model to specify the fog start and fog end distances using view space Z values instead of using 0.0-1.0 normalized device coordinates.

The next section sets the booleans based on the device capabilities. It returns immediately if the device does not support vertex or table fog. If this is the case, our application will not be able to use fog.

```
    // Retrieve supported modes
    bTableSupport  = (Caps.RasterCaps & D3DPRASTERCAPS_FOGTABLE) != 0;
    bVertexSupport = (Caps.RasterCaps & D3DPRASTERCAPS_FOGVERTEX) != 0;
    bRangeSupport  = (Caps.RasterCaps & D3DPRASTERCAPS_FOGRANGE) != 0;
    bWFogSupport   = (Caps.RasterCaps & D3DPRASTERCAPS_WFOG) != 0;

    // No fog supported at all (silently return)?
    if ( !bTableSupport && !bVertexSupport ) return true;
```

Next we perform a few simple comparisons. If the passed TerrainEntity structure had the FOG_PREFERTABLEFOG flag set but the device does not support table fog, we will un-set this flag.

```
    // Strip table flag if not supported
    if ( !bTableSupport ) m_FogEntity.Flags &= ~FOG_PREFERTABLEFOG;
```

If vertex fog is not supported, then table fog must be supported or we would have returned from the function already. Therefore, we make sure that the FOG_PREFERTABLEFOG flag is set. This flag will only be forcefully set if vertex fog is not supported. If vertex fog is supported and the level creator did not specify the preference for table fog, vertex fog will be used.

```
    // Ensure use of table if vertex not supported
    if ( !bVertexSupport ) m_FogEntity.Flags |= FOG_PREFERTABLEFOG;
```

At this point, we have modified the TerrainEntity structure's Flags member such that we know whether we are going to use table fog or vertex fog. In table fog mode we have something else to worry about. If WFog is not supported, we will need to adjust the Fog Start and Fog End members of the terrain entity structure so that they are specified in normalized device coordinates instead of as view space Z values.

The next section of code is executed if table fog is going to be used.

```
    // Fog mode specific setup
    if ( m_FogEntity.Flags & FOG_PREFERTABLEFOG )
    {
        // Ranged is never supported in table mode, strip it
        m_FogEntity.Flags &= ~FOG_USERANGEFOG;

        // If WFog is not supported, we have to adjust these values
         // to ensure that they
        // are in the 0 - 1.0f range.
        if ( !bWFogSupport )
        {
            float fNear = pApp->GetNearPlaneDistance();
            float fFar  = pApp->GetFarPlaneDistance();

            // Shift into range between near / far plane
            m_FogEntity.FogStart -= fNear;
            m_FogEntity.FogEnd   -= fNear;
            m_FogEntity.FogStart /= fFar;
```

```
            m_FogEntity.FogEnd   /= fFar;

        } // End if no WFog support

    } // End if using table mode
```

If the FOG_USERANGEFOG flag is set in the terrain entity structure we remove it since range fogging is only applicable to vertex fog mode. We then test to see if W fog is supported. If it is, then we have nothing else to do since the Fog Start and Fog End members in the terrain entity structure will already be defined in view space (exactly how we should pass them to the pipeline). If it is not supported, we need to map these values into the [0.0, 1.0] range between the near and far planes. We do this by retrieving the near and far plane values currently being used by the CGameApp class via two of its member functions. We then subtract the near plane from the start and end values and divide them by the far plane value. This results in fog start and end values in the [0.0, 1.0] range that describe depth values between those two planes.

Finally if vertex fog is being used instead of table fog, we test to see if the range fog is supported and if not, we make sure the range fog flag in the terrain entity structure is unset.

```
    else
    {
        // If ranged is not supported, strip it
        if ( !bRangeSupport ) m_FogEntity.Flags &= ~FOG_USERANGEFOG;

    } // End if using vertex mode

    // Enable fog
    m_bFogEnabled = true;

    // Success!
    return true;
}
```

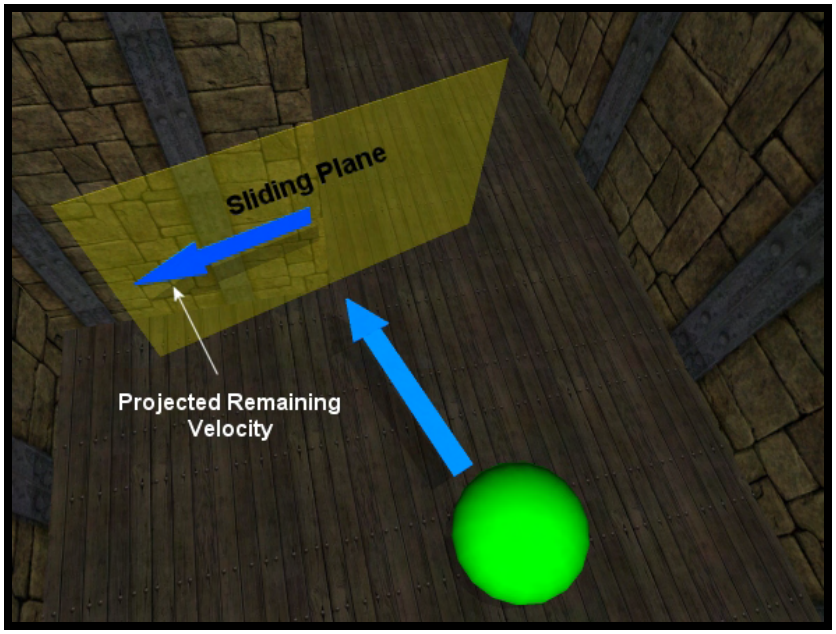
At the end of the function we set the scene's m_bFogEnabled boolean to true so the render function knows to use fog.

Conclusion

A good deal of this workbook has been about consolidating much of what we have learned over the series and upgrading our application framework to support some more advanced constructs. We now have an easy way to describe complex animation sequences for our actor and we have upgraded our code to support multiple terrains, fog, animated actors, trees, and meshes all within the same scene. This is a good place to end the first half of this course because for the remainder of this course, our focus is going to shift almost entirely into the realm of engine design topics. While the goal will not be to design our final game engine just yet (that is our job in Module III), it will be to lay a foundation to do exactly that. We have a lot of extremely important generic game programming topics we need to get through in the coming weeks, so make sure that you feel comfortable with everything we have covered to date.

Chapter Twelve

Collision Detection and Response



Introduction

Since the first lesson in this series we have been implementing visualization applications that permit the camera and other entities to move through solid objects in the game world. But realism in games requires a sense of physical presence and adherence to some simple laws of physics in addition to the visual experience. In the real world, two objects should not be able to simultaneously share the same location. Doing so would indicate that the objects are interpenetrating one another. Since this is not a valid real world state, it cannot be a valid game state either if we want the player to feel as though they inhabit a physical and interactive game space. What our application needs is a means for determining when this situation occurs as well as a means for doing something about it when it does. These two roles fall into the aptly named categories: collision detection and collision response, respectively.

Collision detection and response is actually one of the more heavily researched areas in academic and industrial computer science because there are so many beneficial applications. For example, automobile manufacturers can develop simulations that model various impact phenomena and apply what they learn to improve safety. Of course, as game programmers, we obviously do not have to worry about life and death collision issues in our simulations, and thus our game requirements will be far less demanding. Indeed we will even allow ourselves to occasionally bend or even break the laws of physics and do things that would be virtually impossible in the real world because the results just look better on screen.

More importantly, given the steady increase of geometric complexity in 3D models and environments, it is critical that fast and precise collision determinations and corrections can be made for real-time applications to be able to benefit. Not surprisingly, speed and precision are inversely proportional concepts. The more precise we want our simulation to be, the longer it will take to perform the calculations. Testing every polygon of every object against every polygon of every other object will certainly provide a good amount of precision, but it will be expensive. Modeling proper physics equations in our response engine will generate very precise results, but it will not be as quick as using decent looking (or feeling) approximations. What is required is an appropriate degree of balance between our need for speed and our desire for accuracy.

There are a number of collision systems available to the game developer. Some are free and others are available for purchase and/or licensing. Some systems focus exclusively on the detection portion (SOLID™, I-COLLIDE™, V-COLLIDE™, etc) and require the developer to implement the response phase, while others will include the response side as well (rigid body physics libraries like Havok™ fall into this category and they often include lots of wonderful extra features). The system we will implement in this chapter will include both collision detection and collision response.

This is going to be challenging subject matter, but it is a system that involves concepts that every game programmer should be familiar with. Many of you may even have to implement a system like this at some point in your career. It is probably fair to say that, given the complexity of such systems, many of you may find this chapter to be the most demanding in the training series so far. This is due in large part to a heavy reliance on a significant number of mathematical formulas that must be understood to achieve accurate intersection determination. Rest assured though that we will make every effort in this chapter to try to leave no stone unturned. Hopefully you will find that when all is said and done, it was not so bad after all.

12.1 Our Collision System (Overview)

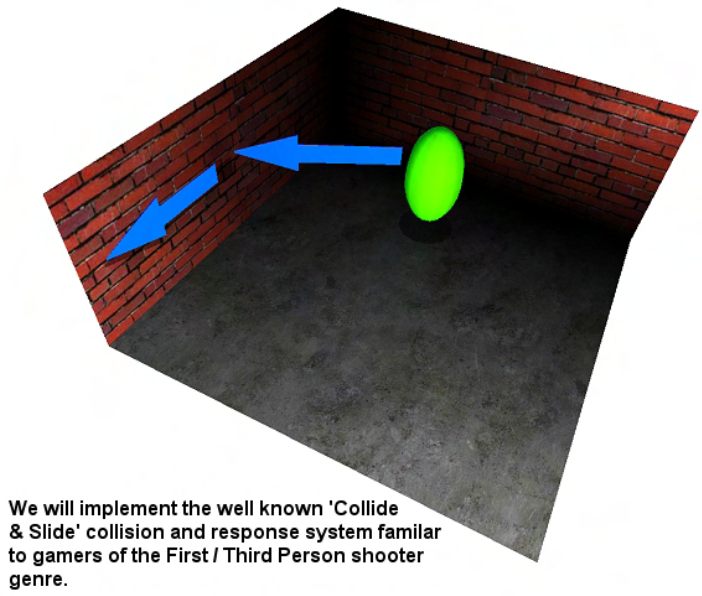
To more accurately model a physical environment, if the user wishes to move in a certain direction at a certain speed, our input code can no longer simply obey this request as it did before. This would allow the player to move through solid objects or embed himself in walls, floors, etc. The same would hold true for other moving objects. Therefore, before any moving entity in our game world can set its final position, it must feed the collision system with information about where it is currently located and where it would like to go. The collision system is tasked with determining whether a clear and unobstructed path is available. If there is a path, then it will let the caller know that it can move the entity to the desired location. If the path is blocked by one or more polygons, then the collision system will calculate and return a final position for the entity that will be different than the requested destination. Suffice to say that the moving entity ultimately ends up deferring to the collision system when it comes to navigating around in the game world. When a clear path is not available, the collision system will essentially say to the entity, "You cannot move where you wanted to move because something was in the way. However, you can move to this other location instead, which has been tested and found to be free from obstruction". Ultimately, the collision detection and response system is responsible for calculating positions that are guaranteed not to be embedded in any solid geometry that comprises the scene.

So our task in this chapter is to figure out a way to prevent moving objects from passing through the polygons that comprise our environment – easier said than done, of course. In Lab Project 12.1 we will use the camera (or more correctly, its attached CPlayer object) as the moving entity we wish to navigate through the level with simulated collision detection and response. We will also have the ability to switch to a third person mode where we will see a skinned character using the collision system in the same way. This will demonstrate that our collision system can be used by any moving entities in the game world to achieve the desired results.

To further clarify the interaction between the moving entity and the collision system we will implement in this chapter, imagine that our moving entity is an ellipsoid and that we wish to move it forward from its current position by 25 units along the world Z axis (velocity vector V = <0, 0, 25>). The entity's current position and its desired velocity vector would be fed into the collision system as a request for a position update. The collision system would then check whether any of the game world polygons would block the displacement of the ellipsoid along this path. For example, let us assume that there is a wall at a distance of 10 units in front of our moving entity. In such a case, our object should only be allowed to move as far as the wall with respect to traveling along its velocity vector (a distance of 10 units).

The collision detection system would initially detect that a collision would occur with the wall. The position that it gives back would be one that corresponded to the entity being just in contact with the wall (i.e., it is moved as far as it can go in the direction described by its velocity vector). Indeed, given this bit of knowledge, we could even say that the collision system *prevents* interpenetration from ever happening. But what about the remaining velocity? We originally wanted to travel 25 units forward, but only managed to travel 10 units before a collision was determined and additional forward movement was prevented. So in a sense, we have an 'energy surplus' that could theoretically carry us for another 15 units of travel. How we handle this remaining velocity is very much a function of the type of collision response system we decide to implement (i.e., our game physics).

In this chapter, the response system we implement will model those that are typically seen in 3D computer games of the 1st/3rd person genre. When a collision is detected, the moving entity will not simply stop at the obstructing polygon and have its excess velocity discarded. Instead, it will slide around such obstructions where possible and use up its remaining velocity. Such a response system will allow us to slide along walls, go up and down stairs and ramps, and smoothly glide over recesses in walls and floors. This is a very common response system in games because it maintains a sense of fluid movement and prevents the player (or other NPCs) from getting stuck on surfaces and edges during fast-paced gameplay.



We will implement the well known 'Collide & Slide' collision and response system familar to gamers of the First / Third Person shooter genre.

Figure 12.1

Our collision system will consist of a *detection phase* and a *response phase*. Both will be called either iteratively or recursively until the energy contained in the initial velocity vector has been completely spent and the entity rests in its final non-penetrating position (more on this later). When a request is made to the collision system to update the position of a moving entity and a collision occurs, the detection phase will determine a position that is flush against the closest colliding polygon (a wall for example). The system response phase will then project any remaining velocity onto what we call the *slide plane*. In the simplest collision cases, the slide plane is simply the plane of the polygon that was collided with (the wall plane, using our current example). There are more complex cases where the moving entity collides with the edge of a polygon (such as on an external corner where two polygons meet or on the edge of a step) where the slide plane will be determined independently from the plane of the collision polygon, and we will discuss such cases later. In the end, this system will allow the moving entity to slide up or down those steps and around such corners.

In Figure 12.1 we see the simple case where the movement of an ellipsoid would cause a collision with a wall. The ellipsoid could only be moved as far as the wall given the information returned by the detection step. Any remaining velocity would be projected onto the slide plane (the wall polygon) in the first iteration of the response step. This results in yet another velocity vector (called the *slide vector*) along which to move the ellipsoid during the next iteration of the system. Moving the ellipsoid along this new velocity vector would result in the appearance of our ellipsoid sliding along the wall.

So we now know that the information returned from the collision step allows us to move the entity into a guaranteed unobstructed position along the velocity vector, and we will spend a lot of time later in this chapter learning how the collision detection phase does this. We also know that the response portion of our system will then project any remaining velocity onto the potential sliding plane to create a new velocity vector along which to slide (starting from the new position determined by the detection phase results). If the detection phase was able to determine that the entity could move to its requested position without a collision occurring, then there will obviously be no remaining velocity for the response phase

to handle. When this is the case, the collision system can simply return immediately and notify the moving entity that it can indeed move into its requested position along the velocity vector. Likewise, even if a collision did occur, if the remaining velocity that is projected onto the sliding plane is so small as to be considered negligible, then we can also return from the collision system with the new position that has been calculated. Remember, the detection phase will always calculate the closest position such that the entity is *not* interpenetrating (or passing through) any geometry. So even if we hit a wall head-on and the projected remaining velocity was zero, we can still return this new position back to the application so that it can update the old position of the entity to this new position (e.g., snug against a wall).

Not surprisingly, the real challenge occurs when the response step successfully projects the remaining velocity vector onto the sliding plane and the result is a slide vector of significant length. In these cases, we find ourselves with an iterative situation. The detection step detects an intersection along the original velocity vector and we calculate a new position such that the entity is moved as far along that vector as possible until the moment the collision occurred. The response step then projects any remaining velocity onto the sliding plane to create a new velocity vector. So in effect, the true position of the entity after the collision should be calculated by adding the new projected velocity (calculated in the response step) to the new position of the entity (calculated by the detection step). Imagine the simple case of an entity hitting a wall at an angle. The collision detection phase would move the entity up as far as the wall and return this new non-intersecting position. The response step would then project any remaining velocity onto the wall to create a new velocity vector pointing along the wall, describing the sliding direction. Therefore, the correct position of the entity after collision processing should be the combination of moving it up to the wall (detection phase) and then sliding it sideways along the wall using its new projected velocity vector (response phase).

> **Note:** It is important to make clear at this point that the terminology and descriptions we are using here are for illustration only. For example, the detection phase does not actually "move" anything. What it does, taking into account the entity's original position, velocity, and bounding volume, is determine the point beyond which the entity cannot go because something in the environment blocks it. It does not physically move the entity, but rather simulates what would happen if we did move the entity according to the requested velocity. However, for the purposes of our current high level discussion, it can be helpful to think of the breakdown of responsibility between the two system phases in the way just described. Later on we will see the precise inner workings of both system components and the details will become clearer to you.

At first, what we just described does not sound like a problem at all. After all, we have the new position and the new projected velocity vector that we have just calculated, so surely we can just do:

Final Position = New Position + New Velocity

In this case, New Position is the guaranteed non-intersecting position of the entity along the original velocity vector calculated during the detection phase and New Velocity is the remainder of the original velocity vector projected onto the sliding plane, starting from New Position.

While this is true, the problem is that we do not know whether the path to this Final Position is free from obstruction. When the collision detection phase was first invoked, it was searching for intersections along the original velocity vector (which it correctly determined and returned a new position for). But

certainly we cannot simply slide the entity along the projected velocity vector calculated by the response step without knowing that this path is also free from obstruction.

As you might have guessed, our solution will need to involve either a recursive or iterative process. If the projection of the remaining velocity vector onto the sliding plane is not zero, we must feed our new position (returned from the detection phase) and our new velocity vector (calculated in the response phase) into the detection phase all over again. The detection phase will then determine any intersections along the new velocity vector and calculate a new position for the entity. This is passed on to the response step and the whole process repeats (potentially many times). It is only when the entity has been successfully moved into a position without any intersections or there is no remaining velocity left to spend beyond the point of intersection, that we can be satisfied that our task is complete. Then we can return a final resting position for our moving entity that we are confident is free from obstruction.

Given the iterative nature of the system, performance is obviously going to be a very real concern. This is especially true when we consider that the detection phase will theoretically need to check every polygon in the scene every time it is invoked in order to accurately determine unobstructed locations. Given the polygon budget for current commercial game environments, testing every scene polygon during every call to the detection phase is not an acceptable design. As a result, commercial collision systems will implement some manner of hierarchical spatial partitioning that encapsulates scene polygons into more manageable datasets for the purpose of faster queries. Indeed, once we have a working collision detection and response system of our own, we will integrate our own spatial partitioning technique (a quad-tree, oct-tree, kD-tree, etc.) to reduce the number polygons that need to be tested for intersection when the detection phase is invoked. We will be examining spatial partitioning techniques and data structures in the next chapter with an eye towards speeding up the detection phase. For now though, we will forego any concerns about optimization and concentrate solely on the core system components and how they work.

Before moving on, let us summarize some of the ideas presented:

1. The collision system will be invoked by the application whenever it wishes to move an object.
2. The collision system inputs will be the current position of the moving entity and a velocity vector describing a direction and distance over which to move the entity. We will also want to input information describing the shape/volume of the entity.
3. The position and velocity are initially passed into the **collision detection phase** of the collision system. This goal of this phase is finding the closest colliding polygon. If no such polygon exists, the system can return the sum of the position and velocity vectors. If a colliding polygon is found along the path of the moving entity:
    a. Calculate and return a new position that it is as far as possible along the input velocity vector, stopping at or before the point where an intersection would occur.
    b. Calculate and return a normal at the point of intersection. In our system, this will act as the normal of a plane that the entity should slide along if there is any velocity remaining beyond the point of intersection.
4. The updated position, the remaining velocity, and the slide plane normal (i.e. intersection normal) are input into the **collision response phase**. The goal in this phase is to apply whatever physics are needed to calculate the contact response of the entity. In our system, we will project the remaining velocity vector onto a sliding plane to generate a new velocity vector that points in

the direction that the entity should slide. We will then pass the updated position and slide velocity vector back into the system to begin the process anew.

Note: Steps 3a and 3b above are not always necessarily going to be tasks assigned to the collision detection phase in every collision system on the market. In fact, most collision detection systems literally just tell you the polygon(s) intersected, and both when and where those intersections occurred. Our detection phase will do a bit more internal housekeeping to make things a little easier to process. Theoretically, these jobs could just as easily have been assigned to the response phase of the system. How you decide to do it in your own implementations is totally up to you.

Later we will learn that the projection of the slide vector will be scaled by the angle between the original velocity vector and the normal of the sliding plane. This way, if we collide with a wall at a very shallow angle, much of the remaining velocity will survive the projection onto the sliding plane and our moving entity will slide along the wall quite quickly. However, if we collide with the wall at a very steep angle (almost perpendicular) then we are hitting the object virtually head-on and the projected velocity vector will be very small, if not zero. In this case we significantly reduce all forward movement and do not slide very much (if at all). This is the response behavior that is implemented in most commercial first-person shooter games.

Figure 12.2 demonstrates the dependency between the projection of the original velocity vector onto the sliding plane and the angle between the original velocity vector and the slide plane normal. In this image, Source is the position of a sphere prior to the update. (We will assume that the sphere is a bounding volume used to describe our moving entity inside the collision system). This current sphere position is one of the initial inputs to the collision system. The red dotted line describes the original velocity vector. This vector tells the collision system both the direction and distance we wish to move the entity. By adding the velocity vector and the Source vector, we get the Original Dest vector which describes the position we wish to move our sphere to (provided that no collisions occur).

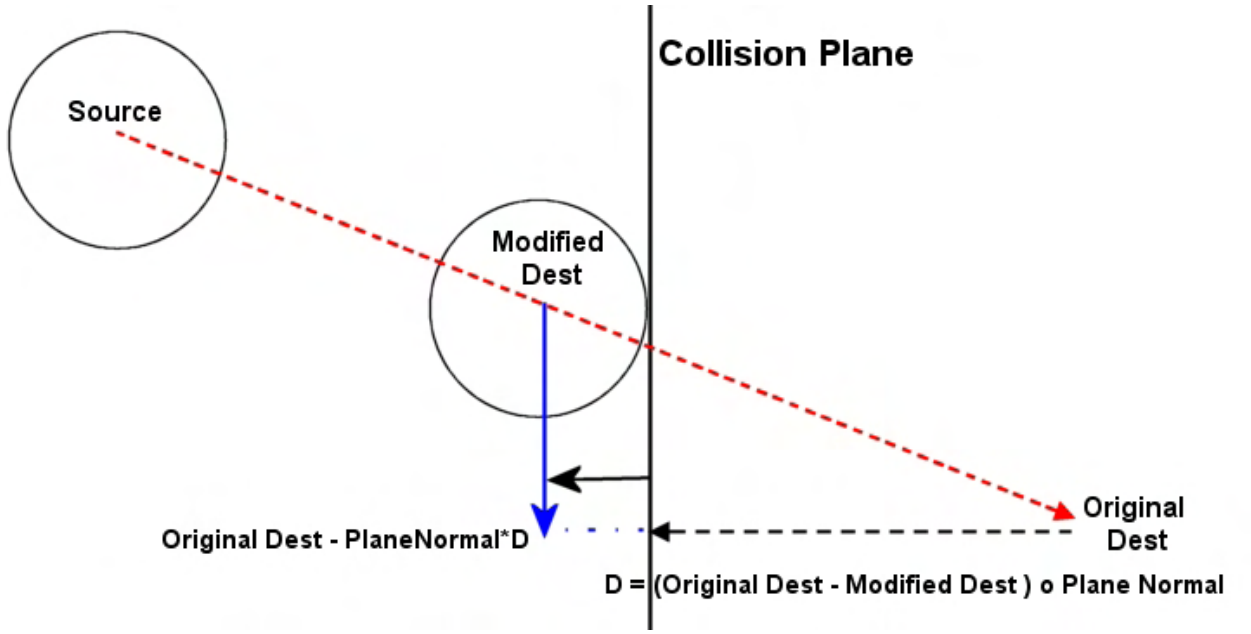


Figure 12.2

We know that the first thing we do is feed this information into the collision detection phase. We are imagining in this instance that a polygon (whose plane is marked as Collision Plane) was found to be blocking the intended path. The detection phase would thus need to return the new sphere position, called Modified Dest, since Original Dest is unreachable. Note that Modified Dest is a sphere position that it is touching, but not intersecting (i.e. does not penetrate), the Collision Plane. We can see clearly that the center of the original sphere only traveled a little less than halfway along the requested velocity vector before hitting the plane, so there is certainly some "leftover" velocity that we can use for sliding. The remaining velocity vector in this example is the vector from Modified Dest (the new center position of the sphere) to Original Dest (the final position of the sphere as originally intended).

When the collision detection phase returns, we have a new sphere position (Modified Dest) and a slide plane normal (the normal of Collision Plane). We can calculate any remaining velocity as:

Remaining Velocity Vector = OriginalDest – Modified Dest

If we perform a dot product between the remaining velocity vector and the unit length plane normal, we get D. D is the length of the adjacent side of a triangle whose hypotenuse is given by the remaining velocity vector (the length of the black dotted line in Figure 12.2). By subtracting D from the Original Dest vector, along the direction of the plane normal, we create the vector shown as the blue arrow in the diagram. This is the projected velocity vector, which forms the opposite side of the triangle. As you can see, this is the direction the sphere should move (i.e., along the plane) to use up the remainder of its velocity. It should also be noted that as the angle of the original velocity vector gets closer to perpendicular with the plane normal, the projected velocity is scaled down more and more until hardly any sliding happens at all (i.e., when the sphere it hitting the plane head-on).

There are times when the slide plane normal returned from the detection routine will not be the same as the normal of the polygon the entity intersected. This is typically the case when the entity collides with a polygon edge (see Figure 12.3).

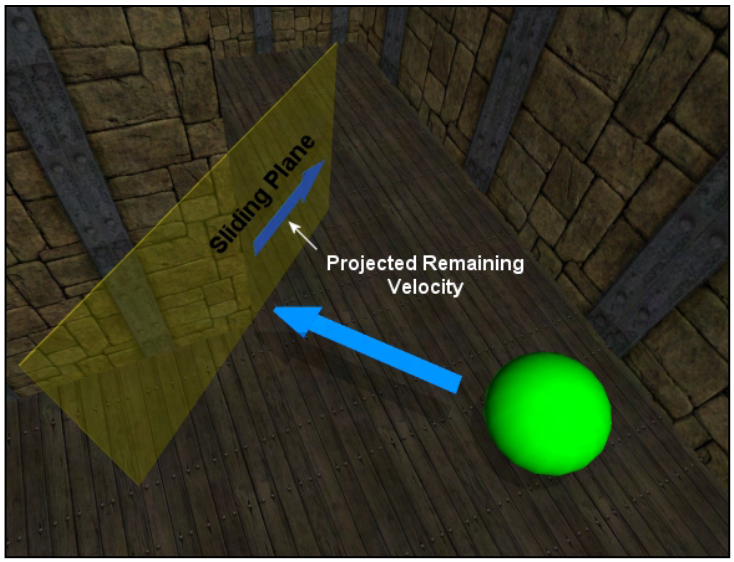


Figure 12.3

In Figure 12.3 we can see that if the green sphere was moved in the direction of the blue arrow, it would collide with the edge where two polygons meet to form an external corner. When this is the case, we want the angle at which the sphere intersects with the edge to determine the orientation of the slide plane. As it happens, this is quite simple to calculate.

When the detection phase calculates the intersection between the edge of a polygon and an entity (a sphere in this example), it will generate the plane normal by building a unit length vector from the intersection point on the edge

to the sphere center point. If you imagine drawing a line from the center of the sphere to the point on the edge where the sphere would hit the corner, you would see that this vector would be perpendicular to the slide plane shown in Figure 12.3. Normalizing this vector gives us the slide plane normal and indeed the slide plane on which to project any remaining velocity that may exist. We can see that in this example, when the sphere hits the edge, it would slide around the corner to the right. Because the slide plane normal returned from an edge intersection is dependant on the position of the entity (the sphere center point in this example), we can see that a different sliding plane would be generated from a collision against the same edge if the sphere used a different angle of approach (see Figures 12.4 and 12.5).

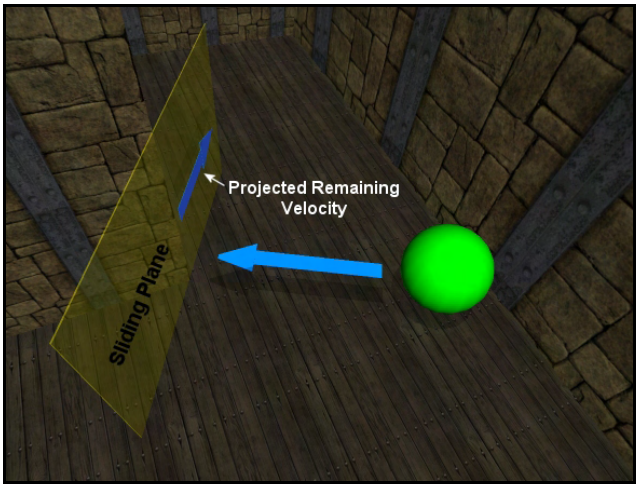


Figure 12.4

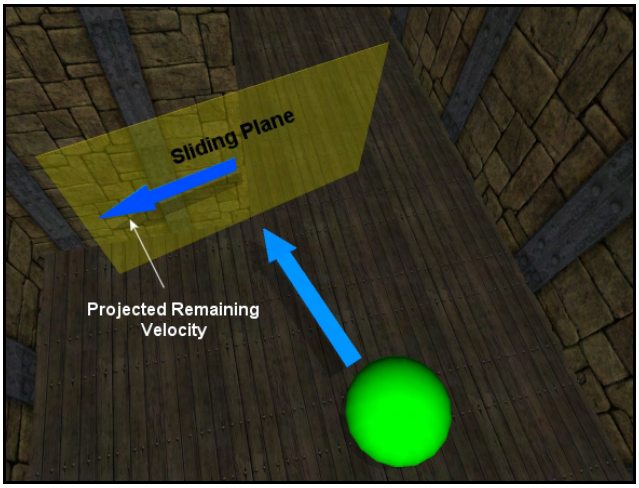


Figure 12.5

It is the sliding plane (and the projection of the remaining velocity vector onto this plane) that also allows the system to automatically handle the navigation of stairs and ramps as well. For example, in Figure 12.6 we see the same sphere attempting to navigate a flight of stairs.

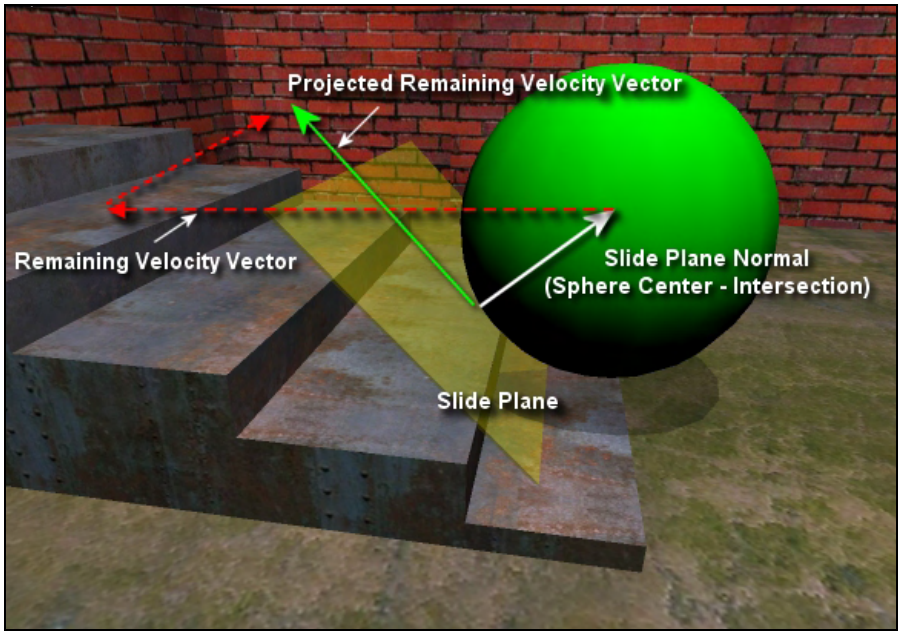


Figure 12.6

In this case, the sphere is already beyond the edge the first step and is trying to travel forward from its current position when it intersects the edge of the second step. The long red dashed line traveling from the sphere center and pointing to the left is the remaining velocity vector after we have first collided with the second step edge.

The small red dashed line shows us the direction of the projection of the remaining velocity vector onto the slide plane.

The collision detection step would determine that somewhere in the bottom left quadrant of the sphere an intersection would happen with the edge of the second step. The slide plane normal would be generated by calculating a vector from this intersection point on the step edge to the sphere center point and normalizing the result. The yellow transparent quad shows the slide plane that would be created for this normal and the plane. This is the plane onto which the response step will project any remaining velocity.

The dot product between the remaining velocity vector (the section of the vector that starts at the new sphere center position calculated by the detection phase) and the slide plane normal creates a new vector tangent to the plane describing the direction in which the sphere should continue to move. This is shown as the green arrow labeled Projected Remaining Velocity Vector. Suffice to say, it is by moving the sphere from its current position (calculated by the detection phase) along this slide vector, that the sphere ends up in its correct final position for the update. We will have moved the sphere as much as possible in the original direction and then diverted all remaining velocity in the slide direction.

As mentioned previously however, the response step cannot simply assume that it can add the slide vector to the position of the new sphere center without testing for obstruction. That is why the response phase must invoke the detection phase again, using the new slide vector as the input velocity vector to the system. This in turn invokes another response phase, and so on. Only when the remaining velocity vector has zero (or near zero) magnitude, do we consider the sphere to be in its final position. Of course, this is always the case as soon as the detection phase determines that it can move the entity along its input vector (which may be the slide vector calculated in a previous response step) without obstruction, since there will be no remaining velocity vector in this instance when the detection phase returns.

Note as well that the height of the step with respect to the entity is very important. The shallower the step, the shallower the slide plane generated. Therefore, any remaining velocity vector will be more strongly preserved when projected onto the sliding plane. For example, in Figure 12.7 we can see the same sphere bounding volume trying to climb a much steeper flight of steps with larger step sizes. The point of intersection with the step is much higher up on the surface of the sphere than before. As such, the slide plane normal calculated from the intersection point to the sphere center position is almost horizontal. In this case, the slide plane is so steep that when any remaining velocity vector is projected onto it, it is almost scaled away to nothing. Depending on your implementation what you might see in

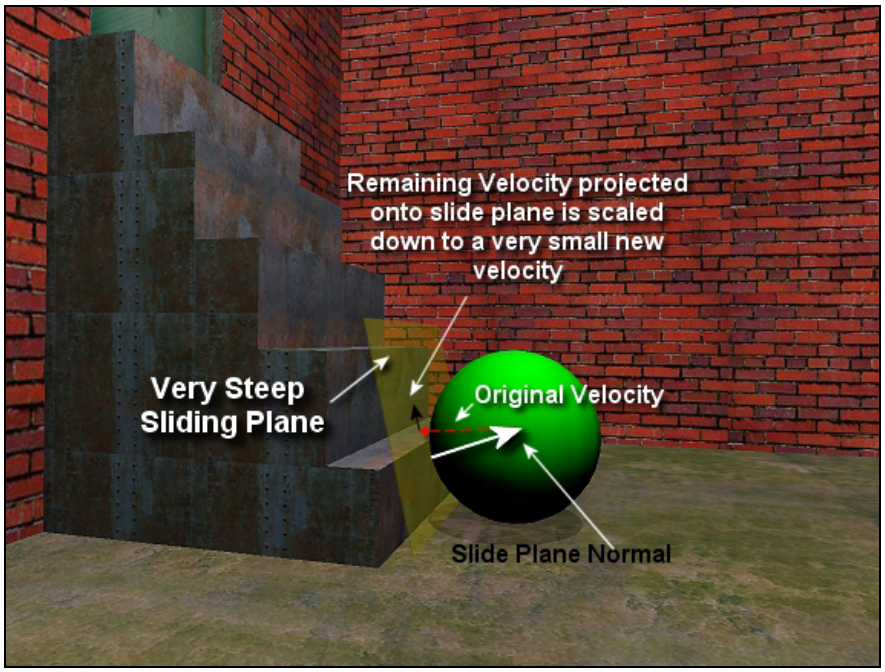


Figure 12.7

this situation is the sphere unsuccessfully attempting to mount the step. Perhaps you would even see a small upwards movement before the sphere came back down again.

Note that this is generally going to be the correct response for this situation. If we imagine the sphere in Figure 12.7 to be the bounding primitive for a character, one step alone would be about half the character's height. We would not expect a character in our game to simply slide over such large obstructions without having to perform some special maneuver (e.g. jumping or climbing). This is an extremely useful side effect of using the sliding plane style of collision response. Indeed it makes the system almost totally self-contained; it will correctly slide over small obstructions (such as steps and debris) and automatically generate very steep sliding planes for larger obstructions, where intersections with the sphere happen a fair way up from its base. Again, the taller the obstruction, the higher the intersection point between the sphere and the polygon will be. As such, the slide plane normal generated by subtracting the intersection point from the sphere center will be almost, if not totally, horizontal. As we have seen, very steep sliding planes zero out any remaining velocity during projection. Therefore, assuming the bounding volume is suitably tall to be consistent with the scale of the geometry, the response step will automatically create shallow sliding planes for collisions against small obstructions like stairs but stop the entity in its tracks when it collides head-on with taller obstructions.

You may wish to verify this behavior now by running Lab Project 12.1. Notice how the camera smoothly glides up stairs and ramps and, with the integration of gravity with the velocity vector, how it falls to the ground when the character walks off a steep ledge. In truth, the gravity vector is not really part of the collision system (it is implemented as part of the CPlayer class movement physics). Every time the position of the CPlayer (which the camera is attached to) is updated, it adds a gravity vector to its current velocity vector. Gravity acts as a constant force, accelerating the entity downwards along the negative world space Y axis during every frame. Since this velocity vector is an input to the collision system (technically, gravity is an acceleration, but we integrate it into a velocity vector for our physics calculations), the entity will automatically fall downwards when there is not a polygon underneath blocking the way. Without the floor of the level for it to collide against, the moving entity would continue to fall forever. While we will talk about this in much more detail later, it demonstrates that from the perspective of the collision system, even the floor of our level is just another polygon to collide and slide against.

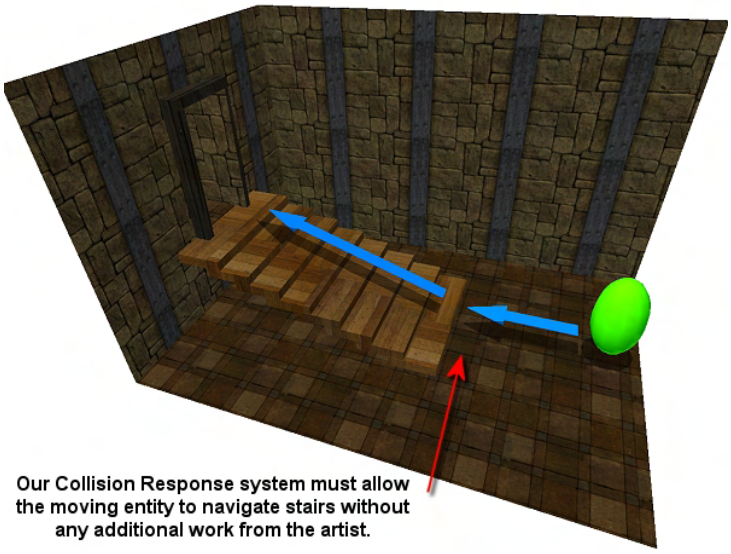


Figure 12.8

As mentioned, stairs and ramps are not the only obstructions that the sliding plane response can handle. It also allows us to slide over small obstructions placed in the game world using the same logic. We certainly would not want the player to be killed in the middle of an intense firefight because his character got stuck while running over a 2-inch thick wood plank. Indeed we want our artists to be able to populate the floor space of our game worlds with lots of interesting details like cracks, dips, splits and

piles of debris; we just do not want these geometric details to prevent us from navigating across the floor (unless they have been designed specifically to be large enough for that task). Our collision system will automatically handle such small obstructions (see Figure 12.9) using the sliding plane response mechanism.

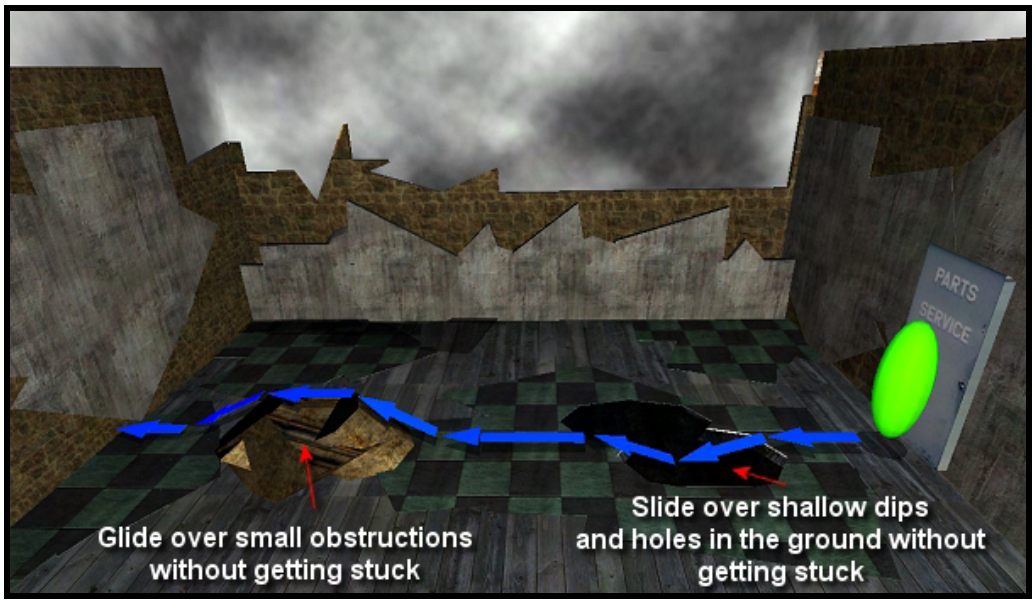


Figure 12.9

The sliding plane response concept will be familiar to gamers who enjoy titles in the first and third person game genres, but it is worth noting that this behavior is easily replaced or augmented according to application requirements. For example, you could replace the response system so that instead of projecting any remaining velocity, you calculate a reflection vector with respect to the collision plane. This would cause the objects to bounce off the polygons in the environment and might be a useful response system when implementing an outer space themed game. Alternatively, the response phase could be simplified so that the position returned from the first iteration of the detection phase of the collision system is used 'as is'. This would mean that when objects collide with the environment, they stop moving. This would feel a little bit like navigating around a world where all polygons are coated with a strong adhesive. While not very useful in most games, it would at least provide a sense of physical presence to your polygons. The larger point here is that your response physics can be as simple or as complicated as you want them to be. As long as you maintain separate detection and response sub-systems, you can configure your results as needed.

In actual practice, the (sliding) response phase we use in our collision system, which features most of the behavior we have discussed so far, can be implemented using only a few lines of code. It involves nothing more than projecting (using a dot product) the remaining velocity vector onto the sliding plane returned from the detection phase and then re-calling the detection phase with new input parameters. Technically, there are a few additional tests we will do in the response phase to reduce certain artifacts. For example, a jittering effect can occur when forcing an entity into an area where two polygons form an internal corner. This jittering results from trying to solve multi-polygon collisions one at a time rather than simultaneously (which is much more computationally expensive). Fortunately, jittering artifacts can be mostly eliminated with another dot product. This will make sure that during any iteration, the

response step does not try to slide the entity in the direction opposite the initial input velocity vector for a given position update. We will cover all of this in detail when we discuss the full implementation later in the chapter. For now, just know that the response phase is quite straightforward and will not require significant effort to implement.

The detection phase is much more complex. If all we had to do was detect was whether or not an entity's bounding volume currently intersects polygons in our scene database, then that would be fairly easy to do (relatively speaking – there is still a good amount of math involved, as we will see later). However, the objects we are concerned with are not standing still, so we have to detect whether the bounding volume will intersect any geometry in the scene *at any time* along its movement vector, before we actually move it. The mathematics for this are a little more involved. Our collision detection phase will also be responsible for returning a new position describing the furthest point the entity can move along that vector before the intersection occurred.

Before we begin our detailed examination of the detection phase and look at the various intersection routines we must implement that make it all possible, we will discuss our high level objectives for the system design and talk about some of the key components that will be necessary for the final system to come together. Note that this chapter is going to be a little different from previous ones in that we will actually be including a good amount of discussion of the lab project source code here in the textbook rather than leaving it all for the workbook. Looking at the source code should help cement a lot of the theory we will be introducing, so keep this in mind as you move forward.

12.2 The Collision System Design

As stated, our primary goal in this chapter is the implementation of a robust and reusable collision system. Along the way, we will progress through the development of a set of class code (CCollision) that can be used by moving entities in our game world to provide accurate collision detection and response. This collision system will be able to be plugged into all of our future applications and can be easily modified and extended to provide different response behaviors.

The collision detection phase will incorporate several static intersection routines. These routines will be included in the system's namespace. Because they are static, these intersection routines can also be called from any module outside the collision system. This will provide future applications with a useful library of intersection functions that can be used with or without the main collision system

> **NOTE:** Because our intersection routines are static and rely on no member variables of the CCollision class, they may be invoked using the CCollision namespace even if a CCollision object has not been instantiated.

Finally, the collision system we develop will place no burden on the project artists to create their geometry in a particular way to make it 'collision compliant'. A primary goal is for the system to work with arbitrarily arranged polygons and meshes (i.e., a polygon soup). This is not to say that you might not want to include artist-placed collidable geometry for particular game purposes (or to solve minor artifact problems should they occur), but that on the whole, the system should be able to work with anything we can throw at it.

12.2.1 Core Components

Our collision system will contain four main components:

Component 1: The Collision Geometry Database

Many tests will need to be made against the geometry of the environment to determine a clear path for a moving entity. Often, the triangles comprising the game world will reside somewhere that would make them either inefficient or impossible to access for collision testing. For example, the vertex data may reside in either local or non-local video memory, or it may have been compressed into a proprietary format that only the driver understands. Having to lock such vertex buffers will cause stalls in the pipeline if the geometry has to be un-swizzled into a format the application can read back and work with. If the buffer is located in video memory, we know that reading back data from an aliased pointer to video memory will result in very serious performance degradation.

Of course, we could create managed vertex buffers (i.e., a resident system memory copy maintained by the D3D memory management system) and lock this buffer with a read-only flag. This would allow our collision system to lock and read back that data without causing a pipeline stall or a possible re-compression step. But this is still far from ideal. When an entity moves, we do not want to have to lock vertex and index buffers and determine which indices in the index buffer and vertices in the vertex buffer form the triangles the entity must collide with. Large scenes would potentially require locking and unlocking many vertex and index buffers, and would prove to be terribly inefficient.

Thus, it is usually the case that a collision system will maintain its own copy of the scene geometry. That data can be stored in an application-friendly format in system memory, making it very efficient for the CPU to access. This is very important because the collision detection will be done by the host CPU and not by the graphics hardware.

Our CCollision class will expose a series of functions that will allow an application to register geometry with the collision system. Therefore our CCollision class will maintain its own **geometry database**. An application loading geometry for scene rendering can just pass the same triangle data into the collision system's geometry registration methods. As our collision system will allow us to pass triangle data to its geometry database as a triangle list, this makes it very convenient to use. Both X files and IWF files export triangle lists, so when our application loads them in, we can simply pass the triangle lists directly into the collision system so that it can store an internal copy for its own use. For convenience, the collision system we develop in this chapter will also allow us to pass in an ID3DXMesh or even a CActor object. It will handle registration of the triangle data contained with these object types automatically. Thus our application could load a single X file containing the main game level (using D3DXLoadMeshFromX for example) and just pass it into the CCollision geometry registration function. The collision system would then have its own copy of the scene geometry and all the information it needs about the environment.

So we will accept that the triangle data used for rendering and the data used for the collision system, while related, are totally independent from one another. While this obviously increases our memory requirements, the collision geometry need not be an exact copy of the triangle data used to render the game world. For starters, we can reduce memory footprint by eliminating unnecessary vertex components like normals and texture coordinates. Additionally, our geometry can be defined at a much coarser resolution to speed up collision queries and to conserve memory. For example, instead of our moving entity having to test collisions with a 2000 triangle sphere, the collision detection geometry version of this sphere may be made up of only 100 polygons approximating a rougher bounding volume around the original geometry. This allows us to speed up collision detection by reducing the number of triangles that need to be tested and it lowers the memory overhead caused by maintaining of an additional copy of the original scene data. If you consider that a wall in your game world might have been highly tessellated by the artist to facilitate some fancy lighting or texturing technique, the collision system's version of that wall could be just a simple quad (eliminating pointless testing). Therefore, while not required by the collision system, the game artists may decide to develop lower polygon versions of the game environment for collision purposes or you may elect to use a mesh simplification algorithm to accomplish the same objective.

One additional benefit can be derived from the separation of rendering and collision geometry: the collision database can contain geometry that the actual renderable scene does not. For example, we can insert invisible walls or ramps in our level simply by adding new polygons only to the collision data. We have all seen this technique employed in games in the past, where the player suddenly hits an invisible wall to stop him wandering out of the action zone.

Component 2: Broad Phase Collision Detection (Bulk Rejection)

The collision detection portion of most systems will distribute the intersection determination load over two stages or phases: a broad phase and a narrow phase. The goal of the **broad phase** is to reduce the number of *potentially* collidable triangles to as few as possible. The goal of the subsequent narrow phase (discussed next) is to test those triangles for intersection and return the results to the response handler.

We simply do not have enough processing power to spare on today's machines to test every polygon in the collision database. While this is obviously true for very large levels, it is also important for small levels (there is no point in wasting CPU cycles unnecessarily). Before intensive calculations are performed at the per-triangle level to determine whether our entity intersects the scene geometry, the broad phase will attempt to (hopefully) eliminate 95% or more of the geometry from consideration. It will accomplish this through the use of any number of different techniques -- hierarchical scene bounding volumes and/or spatial partitioning are the most common solutions. These techniques divide the collision geometry database into some given number of volumes (or 'regions' or 'zones'). We know that if our moving entity has an origin and a destination in only one region, then all other regions (and the polygons stored in those regions) cannot possibly be intersecting our entity along its desired path and require no further testing in the main collision detection phase.

The output of broad phase testing is (hopefully) a very small list of triangles which *might* block the path of the entity, because they essentially 'live in the same neighborhood'. This list is often called the *potential colliders* list as we have been unable to definitively rule out whether or not any of these

triangles are blocking the path of the entity. These triangles (i.e., the potential colliders) become the inputs to the narrow phase where the more time consuming intersection tests are done between the moving entity and each triangle (one triangle at a time).

This component of the system is often separate from the detection and response steps that comprise the core of the collision system discussed earlier in the chapter. In fact, the broad phase may not even be present in a collision system. Although this is the first functional phase in the collision process that is executed whenever a moving entity requires a position update, and indeed it is the one that is most crucial to the performance of our own collision system, this is the phase that we will be adding to our collision system last. In Lab Project 12.1, the broad phase will be absent from our collision system. In fact, it will not be added until the next chapter.

Of course, we would not make such a design decision without good reason. The broad phase we will add to our system in the next chapter will be optionally a quad-tree, an oct-tree, or a kD-tree. These data structures will subdivide space such that the collision geometry database is partitioned into a hierarchy of bounding boxes. Each box is called a *leaf* and will contain some small number of polygons. Because of their hierarchical nature, these trees make per-polygon queries (such as intersection tests) very fast since they allow us to quickly and efficiently reject polygons that could not possibly be intersecting our entity along its desired path.

The broad phase can also be implemented using less complex schemes that rely purely on mesh bounding volume testing (e.g. sphere/sphere, sphere/box, etc.). But even in this case, some manner of hierarchy is preferable. We talked a little bit about such a system design towards the end of Chapter Nine, but we will not give any serious attention on this type of scheme until we examine scene graphs in Module III of this series. Instead we will focus on the highly efficient and very important spatial partition trees just mentioned. Because spatial partitioning is a significant topic all by itself (which needs to be studied in great depth by all game programmers), we have dedicated an entire chapter to it in this course. So we will leave broad phase testing out of the equation for now so as not to lose our focus on the core collision detection and response handling routines.

Component 3: Narrow Phase Collision Detection (Per-Polygon Tests)

In the **narrow phase** of the collision detection step, we will determine the actual intersections with our scene geometry. In our particular system, during the narrow phase we will also attempt to calculate a new non-intersecting position for the entity.

> **NOTE:** It is very important to emphasize that our narrow phase will always return a position that is non-intersecting. This is not always the case in every collision system, but it common enough that we decided to do it in our system as well.

This phase is ultimately responsible for determining if the requested movement along the velocity vector can be achieved without an environment collision taking place. If a collision will occur at some point along the path described by the velocity vector, the narrow phase will determine the maximum position the entity can achieve along that vector before a penetration occurs. It will then return this new position along with the normal at the point of intersection on the polygon (treated as, a sliding plane normal in

our case) to the response phase of the system. This data will be used by the response phase to calculate a slide vector which will be fed in once again to the collision detection system to test for obstruction along the path of the slide vector, and so on. Only when the detection phase is executed without a collision occurring (or if the response phase considers the remaining velocity vector to be of insignificant length) is the iteration stopped and the position returned from the last execution of the detection phase. We can then set the final position of the entity to this new location.

Component 4: Collision Response

The responsibility of our system's **collision response** component is the generation of a new velocity vector (a slide vector) if a collision occurred in the detection phase. This slide vector is computed by projecting any remaining velocity from the previous velocity vector onto a sliding plane returned to the response step by the narrow phase collision detection step. If the projection of the remaining velocity from a previous intersection results in a vector magnitude of insignificant length, then the response step will simply return the position generated during the detection phase back to the application. If a collision occurred in the detection phase and the remaining velocity vector produces a vector of significant length when projected onto the slide plane, the response phase will call the detection phase again with the new position and the slide vector acting as the new velocity vector. The response step will continue this pattern of calling back into the detection phase until no collision occurs or until the length of a projected slide vector is considered insignificant.

The possibility exists with certain types of complex geometry and a large enough initial velocity vector that an expensive series of deep recursions (a slide loop) could occur. In such rare cases, the detection phase would initially detect a collision, the response phase would calculate a new slide vector and pass it to the detection phase, which would detect another collision, and so on. We run the risk of getting mired in a very time consuming recursion waiting for the velocity vector to be completely whittled away. Therefore, our system will put a maximum limit on the number of times the response phase can call the detection phase. If we still have remaining velocity after a maximum number of calls to the detection phase, we will simply discard any remaining velocity and return the non-intersecting position returned from the last detection step. This will all be revisited later when we examine the response step from a coding perspective.

12.2.2 Bounding Volume Selection

In most real-time collision systems, entities are represented using simple bounding volumes like boxes, spheres, cylinders, etc. During broad phase tests, this means that we typically wind up computing very quick volume/volume intersections to reject large areas of our geometry database. Narrow phase collision detection is also often reduced to relatively inexpensive testing for intersections between a bounding volume and a triangle, making this phase suitably efficient for real-time applications.

> **Note:** Many collision detection systems (especially those used in academia) will compute polygon-polygon collisions. These tests are more accurate and thus more expensive. For our purposes, we will generally not require such detailed collision information. Volume-polygon testing is typically all that is

needed for most real-time game development projects, although this is obviously determined by application design requirements.

While there are many different bounding volumes a collision system can use to represent its moving entities, the system we will develop in this course uses ellipsoids as the primary bounding volume. All of our moving entities, regardless of type (a character, a motor car, a bouncing ball), will be assigned a bounding ellipsoid which encompasses the object. When our collision system is asked to calculate the new position of an entity, its ellipsoid will be tested for intersection with the geometry database and used to calculate a new position for that entity. Our decision to use ellipsoids was based on their simplicity of construction and use and their ability to produce a decent fitting bounding volume around the most common types of moving objects we will encounter.

An ellipsoid is the three-dimensional version of an ellipse. It can have three uniquely specified radii along its X, Y and Z extents. Ellipsoids tend to look like stretched or elongated spheres when their radii are not equal. Indeed in the study of quadric surfaces (a family of surfaces to which spheres, cylinders and ellipsoids belong), we find that the sphere is classified as a special form of ellipsoid – one in which the radii along the X, Y and Z extents are equal, giving a single uniform radius about the center point for every point on the sphere's surface.

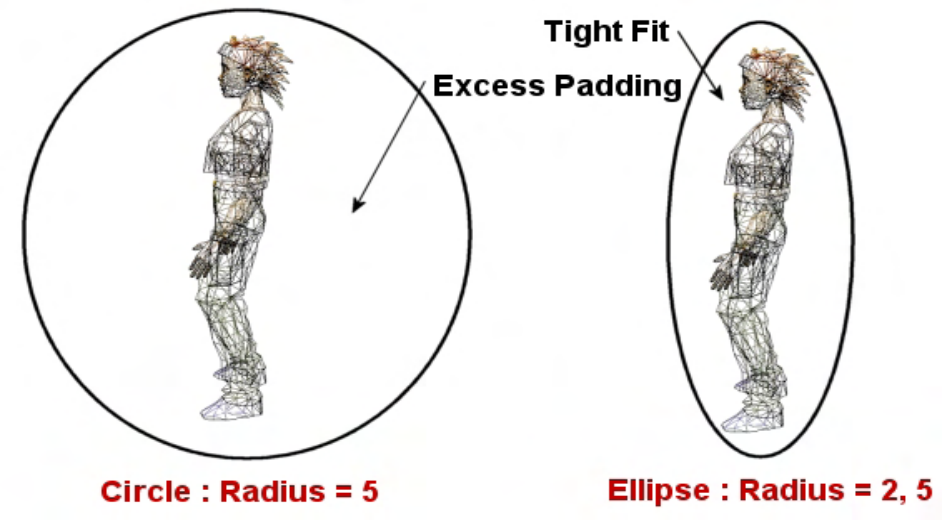


Figure 12.10

We will see later that our collision detection routines will work with unit spheres (spheres with a radius of 1) rather than ellipsoids. Intersection testing is simpler mathematically and more computationally efficient when working with spheres.

However, spheres are not a very friendly bounding volume for most shapes (although they would be a great choice for a beach ball!). If we consider a humanoid character (see Figure 12.10), we can see that if we use a sphere to bound the character, we will get a very loose fit both in front and behind (as well as left and right in the 3D case). This is due to the fact that the sphere's radius has to be sufficiently large to contain the full height of the humanoid form. However, humanoids are generally taller than they are wide, which results in the excess empty space we see depicted. If we imagine the bounding sphere in Figure 12.10 being used by our collision detection system, we would expect that even when the front of the character is quite far away from a polygon (a wall for example), its bounding sphere's surface would intersect it. The character would be prevented from getting any closer to the wall even though it would be clear that adequate room to maneuver exists. The situation is even more problematic when the character wants to walk through a standard sized doorway. The sphere would simply be too wide to accommodate this movement, even when it is clear that the character should easily fit through the opening.

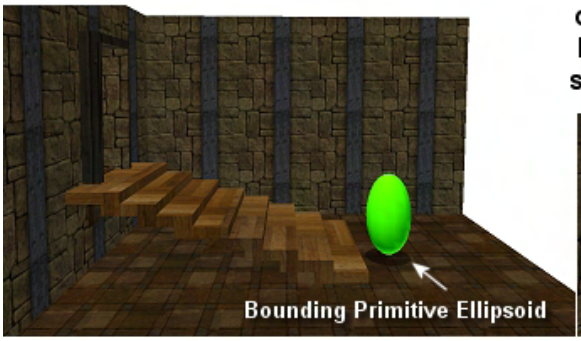
In Figure 12.10 we see that because the height radius can be different than the width/depth radii, we can use an ellipsoid to describe a stretched sphere that better bounds the volume of the entity we are trying to approximate. In this example, the ellipse (2D here for ease of demonstration) has a width radius of 2 and a height radius of 5. An ellipsoid can also be used to tightly bound an animal form (a quadruped) by making the ellipsoid width/depth radii larger than its vertical radius. For example, if you look again at the ellipse shown in figure 12.10 and imagine that the radii were switched so that the ellipse had a width of 5 and a height of 2, we would have a sphere that has been stretched horizontally instead of vertically. This is ideal for bounding a quadruped or even some types of vehicles and aircraft.

So it would seem that from our application's perspective, using the more flexible ellipsoidal bounding primitive for our moving entities is preferable to the use of generic spheres. However, as mentioned, the detection phase would rather use spheres as the bounding primitive to keep the intersection testing mathematics as simple and efficient as possible.

As it happens, we can have the best of both worlds. We can use ellipsoids as the bounding primitives for our collision system's moving entities and, by performing a minor transformation on the scene geometry prior to execution of the detection routines, we can allow our detection routines to use unit sphere mathematics. How does this work?

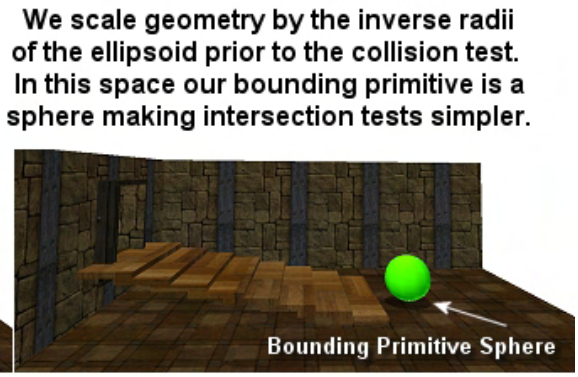
Figure 12.11 shows a small scene and the position of a bounding ellipsoid. In this example, we will assume that the vertices of the geometry and the ellipsoid position are both specified in world space. When we describe the position of the ellipsoid in world space, we are actually describing the world space position of the ellipsoid's center point. In Figure 12.11 we will assume that the ellipsoid has width and depth radii of 1.0 and a height radius of 4.0. Thus, in contrast to a sphere whose radius can be defined by a single value, the extents of the ellipsoid are described by three values (X, Y, and Z radii). They are usually stored in a 3D vector referred to as the radius vector. In our example, the radius vector for the ellipsoid in Figure 12.11 would be:

Radius Vector = (1.0 , 4.0 , 1.0)



Original World Space Geometry

Figure 12.11 : Polygon data in world space



Ellipsoid Space Geometry

Figure 12.12 : Polygon data squashed into ellipsoid space

12.2.3 Inputs/Outputs

The collision system is activated by the application when it wishes to move an object in the scene. This is done with a call to the function shown next. This is the only CCollision method our application will need to call when it wishes to update the position of a moving object.

> **Note:** It should be noted that the *actual* CollideEllipsoid function exposed by our collision system takes a few extra parameters beyond the prototype shown here. These additional parameters will be described later in the chapter when the timing is more appropriate.

```
bool CCollision::CollideEllipsoid (   const D3DXVECTOR3& Center,
                                      const D3DXVECTOR3& Radius,
                                      const D3DXVECTOR3& Velocity,
                                      D3DXVECTOR3& NewCenter,
                                      D3DXVECTOR3& IntegrationVelocity )
```

The function is passed the position of the center of the ellipsoid, the ellipsoid radius vector (describing its three radii) and a velocity vector describing the direction we wish to travel. All three are given in world space units. The length of the velocity vector describes the distance we wish to travel, so we can think of the velocity vector parameter as having both a direction of movement and a desired distance for this particular frame update. The next parameter is a 3D vector which, on function return, will contain the updated position for the ellipsoid. This position will be used to update the position of the entity represented by the ellipsoid. Our final parameter is a 3D vector output parameter called IntegrationVelocity which will contain the 'new' velocity of the object on function return. This final parameter requires a little more explanation.

As discussed, we will pass into the function a velocity that describes the direction and distance we wish to move the object. Our collision system will then repeatedly iterate to move the ellipsoid along this velocity vector until it winds up in a position where no more collisions occur and no velocity remains. We also mentioned earlier that every time the detection phase returns true for an intersection, the remaining velocity (the portion of the velocity vector beyond the point of intersection) is projected onto the slide plane and used to call the detection phase again. This continues until we have spent all the energy in the original velocity vector. It is here that we encounter a problem.

The application will be controlling the velocity of the object using some manner of physics calculations. In our Lab Project for example, when the player presses a movement key, a force is applied to the player. In the player update function, this force is combined with air resistance, drag, and friction to ultimately produce an acceleration which is in turn, integrated into a new velocity vector. This is useful because it means we can apply a large initial force when the user presses a movement key, which results in a vector which can be decelerated to zero over a number of frames (as gravity, drag, etc. operate on the object's mass). When the user releases the movement key, the player will smoothly slow to a halt rather than abruptly stop in their tracks. So what does this have to do with our collision system?

Each time the velocity is calculated in the player's update function, it is later fed into a collision system which whittles it away to nothing during execution. Remember that the system will only return once that velocity has been totally spent. The problem is that our collision system should really not alter the speed

of our player in this way. For example, imagine we were making a space combat simulation where our object (a spaceship) was supposed to be moving through a vacuum. Once the player pressed a key to apply a force/thrust to the ship, even if that key was only pressed for a short time, the object would continue to move at that velocity forever. This is because in a vacuum there are no environmental forces (like resistance and drag) that act to decelerate the object over time. So we know in this case that as soon as the force was applied and the velocity vector was determined to be of a certain magnitude, it should remain at that magnitude for all future frame updates. But if we let our collision system whittle away our actual velocity, every time we called the CollideEllipsoid method for our spaceship, the final velocity would be zero (even if no intersections occur). Since the application needs to apply velocity during every frame update (assuming no forces counteract it) and a zero velocity would clearly have no effect, this is not very helpful.

Now you might think that the obvious solution to the problem is to let the collision system work with a temporary velocity vector and let the object maintain its own copy that is never tampered with by the collision system. But that only works if our object never hits anything. The problem with this approach is that while the collision system should not tamper unnecessarily with the object's speed, it absolutely needs to be able to modify the direction of the velocity vector according to collision events.

Imagine that the player collides with a wall and the collision response step projects the remaining velocity onto the wall and we eventually move the sphere into its final position. At this point the velocity vector will be spent and the collision system can return. However, the collision system needs to let the application know that the object is now traveling in a completely different direction. When the collision system was invoked, the player's velocity vector had a direction that carried it straight into the wall. The system detected this and the response step ultimately modified the direction of travel. Indeed, if several intersections occur in a single movement update, the direction might have to be changed many times before the object comes to rest in its final position. Our collision system will need to return this new direction vector back to the application so that it can use it to overwrite the previous velocity vector and integrate it into physics calculations during future updates. Our collision system will also try to maintain as much of the original input velocity's magnitude as possible. It is this integration velocity vector that is returned from CollideEllipsoid and it represents the new velocity vector for the object.

Calculating the integration velocity is easy enough as it is done in nearly the same way that we project the remaining velocity onto the slide plane. Every time a collision occurs, the response phase receives a slide plane and an intersection point. It uses this information to project the remaining velocity onto the slide plane to generate a new velocity vector as input to the next detection call. All we have to do now is repeat this process during iteration to generate the integration velocity. There is one big difference however. When the integration velocity is projected onto the slide plane, we use the *entire* vector and not just the portion of the vector that lies behind the slide plane. We can think of both the slide plane and the integration velocity as being located at the origin of the coordinate system. A simple direction change is achieved by projection onto the new plane to create a new integration velocity for the next iteration. When the collision detection phase finally returns false (meaning no more responses have to computed), we will have the final integration velocity to return to the application. Note that the continual projection of the integration velocity onto different planes during the iterative process will mean that its length (i.e., speed) will still degrade by some amount. However, when only a few shallow collisions occur (e.g. clipping a corner or sliding into a wall at a small angle), most of the object's original speed will be maintained.

From the application's perspective, using our collision system could not be easier. We just tell it where we are currently situated (Center), how big we are in the world (Radius) and the direction and distance we wish to move from our current position (Velocity). On function return, we will have stored in the parameter (NewCenter) the new position we should move our entity to. Of course, this may be quite different from the position we intended to move to if a collision or multiple collisions occurred. We may have requested a movement forward by 10 units but the detection phase may have detected a collision against a wall. The response phase may have then calculated that it needs to slide to the right by 5 units and called the detection phase again to test that the slide vector path is clear. The detection phase may have discovered another intersection along the slide vector and therefore the position of the ellipsoid may have only been moved partially along the first slide vector. Any un-spent velocity of the slide vector will then be handed back to the response system which will use the remaining velocity of the previous slide vector to calculate a new slide vector from this new polygon that has been hit during the first sliding step. The response system would then call the detection phase again passing in this new (2^{nd}) slide vector and so on until the collision detection finally slides the entity into a position which does not cause an intersection.

12.2.4 Using Collision Ellipsoids

In the last section we learned that our application passes an ellipsoid radius vector into the collision system when it wishes to update the position of an entity in the game world. We know that the detection step will initially use both a broad and narrow phase to determine triangles which may cause obstructions along the path of the velocity vector. Once done, a new step will need to be taken. The narrow phase will transform each triangle it intends to test for intersection into something called *ellipsoid space*.

In ellipsoid space, the ellipsoid is essentially 'squashed' into a unit sphere and the surrounding triangles that are to be tested for intersection are also 'squashed' by that same amount. Therefore, after we have scaled any potentially colliding triangles into ellipsoid space, they now have the same size/ratio relationship with the unit sphere as they did with the ellipsoid before it was transformed. In this manner, our collision system can now perform intersection tests between a unit sphere and the transformed ellipsoid space geometry to efficiently find intersections. Once an intersection is found, the new unit sphere position can be transformed back into world space and returned to the application for final update of the moving entity's position.

To understand this concept of ellipsoid space, take another look at Figure 12.11. We stated earlier that the ellipsoid in this image was assumed to have a radius vector of (1.0, 4.0, 1.0). If we wished to convert this ellipsoid into a unit sphere (a sphere with a radius of 1.0) we would simply divide each component of its radius vector by itself:

$$\textbf{Unit Sphere Radius} = \left(\frac{X}{X}, \frac{Y}{Y}, \frac{Z}{Z}\right)$$

$$= \left(\frac{1.0}{1.0}, \frac{4.0}{4.0}, \frac{1.0}{1.0}\right)$$

$$= (1.0, 1.0, 1.0)$$

$$= 1.0 \text{ Radius}$$

Now this may seem quite obvious because if we divide any number by itself we always get 1.0. However, as can be seen in Figure 12.12, if we scale the X, Y, and Z components of a triangle's vertices by the (reciprocal) components of the ellipsoid's radius vector as well, we essentially squash that triangle into the same space. This is a simple scaling transformation, just like any scaling transform we have encountered in the past. By applying the scale, we wind up bringing both the triangle and ellipsoid into a new shared space (coordinate system).

Note that as far as position and orientation are concerned, we do not consider the ellipsoid center to be the origin of the coordinate system or its radii to be the axes of the system. While it would certainly be possible to transform everything into the local space of the ellipsoid (i.e., use the ellipsoid as a standard frame of reference) it turns out to be an unnecessary step. The world origin and axes can continue to serve as the origin/orientation in this new space, but we will scale everything in that space by the same amount, thus preserving the initial (world) spatial relationships between the ellipsoid and the triangles. In that sense, all we have done here is scale the world coordinate axes using the (reciprocal) ellipsoid radii as our scaling factor. So, ellipsoid space is ultimately just a scaled world space.

In Figure 12.12 we can see that we have scaled the ellipsoid by a factor of 1.0 / 4.0 (its Y radius) along the Y axis and we have also scaled the surrounding geometry (the potential colliders) by the same amount. We can now safely perform intersection tests against the surrounding triangles with a unit sphere since both the triangles and the ellipsoid have been transformed into ellipsoid space. For example, imagine a quad comprising the front of a step that had a height of 6 (its top two vertices had a world space position of 6). We would simply divide the Y components of the vertices of the quad by 4, scaling the height of the step by the same factor that we scaled the height of the ellipsoid to turn it into a sphere. The height of the transformed step quad in ellipsoid space remains just as high proportionally to the unit sphere as the untransformed step quad was to the original ellipsoid.

Therefore, all our collision system will have to do before performing intersection tests between a given triangle and the unit sphere, is divide the components of each of the three triangle vertices by the components of the ellipsoid's radius vector:

```
Ellipsoid Space Position.x =  Center.x / Ellipsoid Radius.x;
Ellipsoid Space Position.y =  Center.y / Ellipsoid Radius.y;
Ellipsoid Space Position.z =  Center.z / Ellipsoid Radius.z;

Ellipsoid Space Velocity.x = Velocity.x / Ellipsoid Radius.x;
Ellipsoid Space Velocity.y = Velocity.y / Ellipsoid Radius.y;
Ellipsoid Space Velocity.z = Velocity.z / Ellipsoid Radius.z;

For ( each vertex in triangle )
{
        Ellipsoid Space Vertex.x = World Space Vertex.x / Ellipsoid Radius.x;
        Ellipsoid Space Vertex.y = World Space Vertex.y / Ellipsoid Radius.y;
        Ellipsoid Space Vertex.z = World Space Vertex.z / Ellipsoid Radius.z;
}
```

> **NOTE:** In practice, we would likely calculate the reciprocal of the ellipsoid radii as a first step and then change all of the divisions into multiplications.

As can be seen in the above pseudo-code, it is very important that the center position of the ellipsoid and the velocity vector passed in by the application be transformed into ellipsoid space using the same method. We need to make sure that all of the positional information about to be used for the intersection testing (entity position, velocity and triangle vertex coordinates), exist in the same space. If we do not do this, the system will not function properly and we will generate incorrect results.

So, when our collision system is invoked by the application to update the position of a moving object, it will transform the input position and velocity vectors into ellipsoid space (from here on in called eSpace). This must take place prior to performing any per-polygon intersection testing. It is worth noting that the system will not use eSpace during the broad phase. Collection of potential triangle colliders will remain a world space operation (we will see this in the next chapter). However, once we have assembled the list of possible colliders, all of them will need their vertex positions transformed into eSpace. We do this by dividing the X, Y and Z components of each vertex by the corresponding radius component of the ellipsoid. The collision system will examine every triangle in the list and test for intersections with a unit sphere at the eSpace input position moving with the eSpace input velocity. Actual intersection positions are recorded along with a parametric distance along the velocity vector indicating when the intersection took place.

> **NOTE:** The collision system cannot simply return after finding the first colliding triangle because the velocity vector may pass through multiple triangles if it is sufficiently long. The detection phase must test all potential colliders and only calculate the new position of the sphere after it has found the *closest intersecting triangle* (if one exists) along the velocity vector. This will be the first triangle that the sphere will hit and must be prevented from passing through.

This should give you some indication as to why the broad phase is so critical to the overall performance of the system. Without some means for quickly rejecting triangles which could not possibly be intersecting the ellipsoid along the velocity vector, our detection routine will have no choice but to test every single triangle in the environment. This will include the added cost of transforming all triangle vertices into eSpace prior to the intersection tests.

Finally, once the collision system is satisfied that it has found the new position, this position will need to be transformed from eSpace back into world space before being returned to the application for entity position update. The transformation of a position vector from eSpace back into world space can be done by simply reversing the transformation we performed earlier. Since dividing the components of a world space vector by the corresponding components of the ellipsoids radius vector transformed it into eSpace, multiplying the components of an eSpace position vector by the components of the ellipsoid's radius vector will transform it from eSpace to world space.

12.2.5 System Summary

We now have a good idea about the type of collision system components we are going to implement and the end results we should expect them to generate. As discussed, the performance enhancing broad phase component will be deferred until the next chapter so that we can focus on the core collision system in our first implementation.

The geometry database component of our CCollision class will be simple in our first implementation and will not require detailed discussion. The CCollision object will maintain an STL vector of triangle structures containing the three triangle indices, the triangle normal, and the material used by the triangle. A material in this case does not refer to a rendering material, but rather to a set of physical properties so that concepts like friction can be calculated. There will also be an STL vector of vertices. This will just be an array of the <X, Y, Z> position components since the collision geometry database has no use for additional vertex components like color or texture coordinates. Our system will provide a simple API for the purposes of adding triangle data to these arrays. The collision detection system will work exclusively with the geometry stored in these vectors, so the application must register any polygons it considers to be 'collidable' before collision testing begins (e.g., when the scene is created/loaded). Please refer to the workbook for details about the functions that add triangles to the collision database.

12.3 Collision Response

The response phase of the collision system is going to be very simple for us to implement; it requires only a few lines of code in its basic form. The collision detection step is where most of our work will occur and it is where we will find ourselves getting into some mathematics that we have not discussed before in this course series. Fortunately, you have likely already encountered most of this math in high school, so it should not be too difficult to follow. Our collision detection routines will do a fair amount of algebraic manipulation and will utilize the quadratic equation quite extensively. Students who may be a little rusty with these concepts need not worry. We have included a very detailed discussion of quadratics and will walk through the algebra one step at a time.

The collision detection phase is where we will focus most of our energies in this first section of the textbook. But before covering the detection phase code in detail, it will be useful to first look at the context in which it is used. We said earlier that the application need only call a single method of the CCollision object in order to update the position of any moving entity. This method is called

CollideEllipsoid. For now, just know that this single function is the heart of our collision system, called by the application to update the position of an entity in the scene. This function iteratively calls the detection phase and executes the response phase until a final position is resolved. We will have a look at this function in a moment and get a real feel for the flow of our system.

The code to the CollideEllipsoid method is quite simple to follow. The caller passes the position vector, radius vector and the velocity of the ellipsoid that is to have its position updated. It then calls a single collision detection function which completely wraps the detection phase of the collision system. This function will return either true on false indicating whether an intersection has happened along the path described by the velocity vector. If no collision has taken place, the CollideEllipsoid function will immediately return the new position to the application. This new position will be Position + Velocity, where both Position and Velocity are the vectors input to the routine by the application.

If an intersection does occur, then the collision detection function (called CCollision::EllipsoidIntersectScene) will return the new position for the object (in a non-intersecting location) and an intersection (slide plane) normal. CollideEllipsoid will then perform the response step and project any remaining velocity onto the slide plane before calling the EllipsoidIntersectScene function again to invoke another detection pass.

Before we look at the code to the CollideEllipsoid function, let us first examine the parameter list to the EllipsoidIntersectScene. Although we will not yet know how the detection phase works, as long as we know the parameters it takes and the values it should return, we will be able to look at the code to the rest of the collision system and understand what is happening.

The collision detection component of our system may sound like a complicated beast, but from a high level it is just a single method that is called by the collision system when a position update is requested. The prototype for the parent function that wraps the entire detection phase is shown below. This function is called by CCollision::CollideEllipsoid (our collision system interface) to determine if the path of the ellipsoid is clear. If not, it returns the furthest position that the ellipsoid can be moved to along the velocity vector such that it is not intersecting any geometry. If this function returns false then no intersection occurred and the entity can be safely moved to the position that was originally requested by the calling application (NewPos = Center + Velocity).

```
bool  CCollision::EllipsoidIntersectScene  (const D3DXVECTOR3& Center,
                                            const D3DXVECTOR3& Radius,
                                            const D3DXVECTOR3& Velocity,
                                            CollIntersect Intersections[],
                                            ULONG & IntersectionCount,
                                            bool bInputEllipsoidSpace = false,
                                            bool bReturnEllipsoidSpace = false );
```

We have quite a lot of work to do before we find out how this function (and the functions it calls) works, but a good place to start is by examining the parameter list and the way that it is used by the collision system in general.

const D3DXVECTOR3& Center
const D3DXVECTOR3& Radius
const D3DXVECTOR3& Velocity
These three parameters tell the detection phase the position of the center point of the ellipsoid we wish to move, its radius vector, and the magnitude and direction in which we wish to move it. The application passes this information into the parent function of the collision system (CCollision::CollideEllipsoid) which passes it straight through to the detection function (EllipsoidIntersectScene) via these parameters.

CollIntersect Intersections[]
The collision system passes in an array of CollIntersect structures. The CollIntersect structure is defined in CCollision.h and is contained in the CCollision namespace. It is the structure that the collision detection phase uses to transport intersection information back to the main collision system (for use in the response phase for example). This information will only be available if a collision occurred. The CollIntersect structure describes the intersection information for a single colliding triangle and is defined as:

```
struct CollIntersect
{
   D3DXVECTOR3 NewCenter;
   D3DXVECTOR3 IntersectPoint;
   D3DXVECTOR3 IntersectNormal;
   float       Interval;
   ULONG       TriangleIndex;
};
```

This structure's members are explained below.

D3DXVECTOR3 NewCenter
This member will contain the new position that the ellipsoid can be moved to without intersecting the triangle. This position can be used in the response phase or it can be passed back to the application (depending on your system's design needs) to update the position of the moving entity which the ellipsoid is approximating. In our system, we will not just return this new position back to the application if there is significant velocity left over beyond the point of intersection. In such cases, the response phase will create a new velocity vector from the remaining velocity by projecting it onto the sliding plane. The response phase will then call the EllipsoidIntersectScene function again to re-run the detection phase and test that the path described by the slide vector is also clear. This variable (NewCenter) will be the new position input as the first parameter in that next iteration of the detection phase. It essentially describes how far we managed to move along the velocity vector fed into the previous invocation of the detection phase.

D3DXVECTOR3 IntersectPoint
This vector will contain the actual point of intersection on the surface of the triangle. This will obviously also be a point on the surface of the ellipsoid (since they are in contact at the same physical point in space), so we now exactly what part of the ellipsoid skin is currently resting against the triangle we intersected with.

D3DXVECTOR3 IntersectNormal

This vector describes the normal at the point of intersection (described above). If the collision detection function determines that the ellipsoid collided somewhere in the interior of the triangle, this vector will simply store the normal of the triangle that was hit. If the ellipsoid collided with the edge of a triangle, then this will be a normal describing the direction to the center of the ellipsoid from the intersection point (returned above). Either way, this is the normal that describes to the response phase the orientation of the plane we wish to slide along if there is any left over velocity beyond the point of intersection. We will treat the IntersectPoint and the IntersectNormal as describing a sliding plane onto which the response phase will project any remaining velocity.

float Interval

This floating point member stores the *t* value for intersection along the velocity vector passed into the detection phase. As you will see in a moment when we start covering the mathematics of intersection, the *t* value describes the time (or distance, depending on context) along the velocity vector when the intersection occurred. The *t* value is in the range [0.0, 1.0] where a value of 0.0 would describe an intersection at the very start of the velocity vector and a value of 1.0 describes an intersection at the very end of the velocity vector. A value of 0.5 describes the collision as happening exactly halfway along the velocity vector. In effect, it describes the time of collision as a parametric percentage value. The point of intersection returned in the IntersectPoint member described above, is calculated as:

```
IntersectPoint = Center + (Velocity * Interval)
```

As our collision detection function does this calculation for us and conveniently returns the intersection point in a separate variable, there is no need for us to do this. But this is how we would do it if we were required to calculate the intersection point ourselves.

ULONG TriangleIndex

This value will store the index of the triangle in the collision system's triangle array (STL vector) that the collision has occurred with. This is often a useful piece of information to return to the caller, so we will include it in our structure.

ULONG & IntersectionCount

You will notice that our collision system passes an array of CollIntersect structures into the EllipsoidIntersectScene function to be filled with intersection information. This is because it is possible that when the new position is calculated, it is actually touching two or more triangles. This may seem strange at first when we consider that we are only interested in finding the first (i.e., closest) intersection time along the velocity vector, and indeed, our collision detection function will only return collision information for the closest impact. Ordinarily this is going to be a single triangle and only the first element in the passed CollIntersect array (Intersections[0]) will contain meaningful values on function return. However, it is possible that the ellipsoid could come into contact with multiple triangles at exactly the same time (see Figure 12.13).

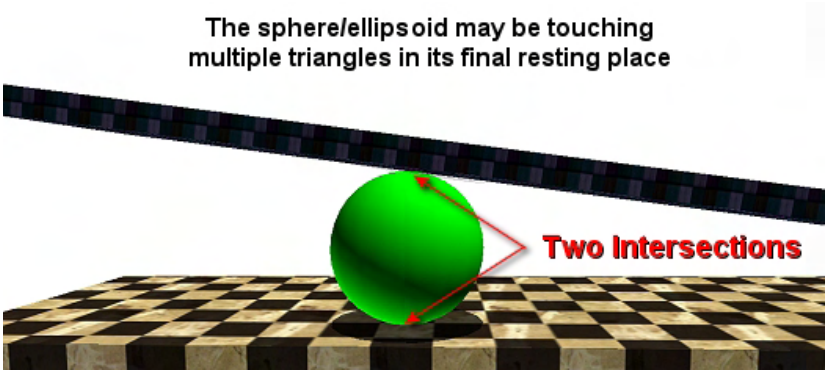


Figure 12.13

As you can see in this image, the sphere is being moved into a wedge shaped geometry arrangement and as such, will intersect with both the floor and ceiling at exactly the same time. In such a case, the detection function would return two pieces of intersection information which have exactly the same time interval. That is, collision information will be stored in the first two elements of the array (Intersections[0] and Intersections[1]) describing the information discussed previously (triangle index, intersection point, etc.). Remember, this will only happen when both intersections happen at exactly the same time along the velocity vector since our collision detection function is only interested in returning collision information for the first time of impact.

The IntersectionCount parameter will contain the number of simultaneous intersections that have occurred. This describes the number of triangles that the detection phase has determined the ellipsoid will be touching in its final position. It obviously also describes the number of elements in the Intersections array that was passed into the detection function.

One thing might be concerning you at this point. When we discussed the CollIntersect structure, we said that it contains a variable called NewCenter which describes to the response step (and possibly the application) the new non-intersecting position of the ellipsoid along the velocity vector. If we have multiple CollIntersect structures returned, which one's NewCenter member should we use to update the position of the entity? As it happens, it does not matter, because our detection function will only return information for the closest intersection. If that intersection involves multiple triangles, the NewCenter vector in each structure will be exactly the same. If you take a look at Figure 12.13 you can see that whether the collision information for the floor or the ceiling is stored first in the intersection array, the same center position for the sphere will be contained in both items. This is still the furthest the sphere can move in the desired direction before the intersections occur. Therefore, it is the only position that can be returned for all intersections that may be happening at that first point(s) of contact along the velocity vector. So while the intersection points and normals will obviously be different, the sphere center position will always be the same amongst simultaneous colliders.

Thus, when the detection phase returns, we can always use the new position stored in the first intersection structure in the array for our response step. The detection phase returns the array of intersection information mostly out of courtesy. Our particular system implementation only needs the new position, so we need not return information about all intersections. However, this might be useful if you intend to extend the collision system later on. After all, we can certainly imagine a situation when it might be useful to know not only if a collision occurred, but have a list of the actual triangles that were hit (perhaps to apply a damage texture to them for example).

Now that we know what the EllipsoidIntersectScene function will expect as input and return as output, we can now examine the code to a CCollision::CollideEllipsoid function that implements the system we have discussed thus far. Remember, this is the only function that needs to be called by the application in

order to update the position of a moving entity. It is also the function that contains all the code for the response step. The only code we will not see just yet is the code to the detection phase. This is tucked away inside the EllipsoidIntersectScene function whose parameter list we have just discussed. Understanding this function will require some study, but now we know how the detection function is expected to work with the rest of the system, so we can understand how everything else in the system works.

The CCollision::CollideEllipsoid function is shown below. This is a slightly more basic version than the one we will finally implement in the workbook. It has had all application specific tweaks removed so that we can concentrate on the core system. The function we are about to examine is really the entire collision system. We should understand everything about our system after studying this code except for how the actual detection phase determines intersections along the velocity vector. This will be discussed in a lot of detail later on in the chapter.

In this first section of code we are reminded that this function is called by the application and passed the position of the ellipsoid that is to be moved, the radius vector, and the velocity vector describing both the direction and distance of movement we desire. At the moment, all three use world space units. For the final parameters, the application passes a reference to two 3D vectors which on function return will contain the new position the application should move the entity to and its new velocity.

```
bool CCollision::CollideEllipsoid ( const D3DXVECTOR3& Center,
                                    const D3DXVECTOR3& Radius,
                                    const D3DXVECTOR3& Velocity,
                                    D3DXVECTOR3& NewCenter,
                                    D3DXVECTOR3& IntegrationVelocity )
{
    D3DXVECTOR3 vecOutputPos, vecOutputVelocity, InvRadius, vecNormal
    D3DXVECTOR3 eVelocity, eInputVelocity, eFrom, eTo;
    ULONG       IntersectionCount, i;
    float       fDistance;
    bool        bHit = false;

    vecOutputPos       = Center + Velocity;
    vecOutputVelocity  = Velocity;

    // Store ellipsoid transformation values
    InvRadius = D3DXVECTOR3( 1.0f / Radius.x, 1.0f / Radius.y, 1.0f / Radius.z );
```

The first thing we do is add the velocity vector to the position passed in and store the result in the vecOutputPos local variable. If no intersections occur then this describes the new position that should be returned to the application. We calculate it here so that if no intersections occur we can simply return it without any additional work. If intersections do occur then this value will be overwritten with the final position determined by the detection and response steps. We also copy the input velocity vector into the local variable eOutputVelocity. This vector will be used to store the integration velocity that will be returned to the application.

Next we create a vector called InvRadius where each component equals the reciprocal of the matching component in the ellipsoid's radius vector. We will need to convert the information (Center, Velocity,

etc.) into eSpace by dividing each vector component by the matching component in the radius vector. Because we will need to do this quite a bit (especially in the detection phase where it will be done for each triangle vertex), we create a vector that when component multiplied with another vector will perform this division. Remember, A / B = A * (1 / B) and therefore, if we create a vector that has the components, 1/Radius.x, 1/Radius.y, and 1/Radius.z, we have a vector that scales by the radius of the ellipsoid to transform points into eSpace.

In the next section of code we will see this transformation being done. As discussed, the detection phase wants to work in eSpace so we must send it the ellipsoid position and its velocity in eSpace, not world space. We accomplish this by multiplying the world space vectors (Velocity and Center) by the inverse radius vector we just computed:

```
    // Calculate values in ellipsoid space
    eVelocity = Vec3VecScale( Velocity, InvRadius );
    eFrom     = Vec3VecScale( Center, InvRadius );
    eTo       = eFrom + eVelocity;

    // Store the input velocity for jump testing
    eInputVelocity = eVelocity;
```

Notice that we store the eSpace velocity vector in eVelocity and the starting position of the ellipsoid in eSpace in the eFrom vector. We also calculate the eSpace position we would like to move the sphere to by adding eVelocity to the eFrom point. If no collisions occur, this is exactly where we would like our sphere to end up in eSpace after travelling the full extent of the velocity vector. We need to know this position in order to calculate any left over velocity in the event of an intersection. This remaining velocity can then be projected onto the slide plane as will be seen in a moment. A copy of the initial velocity vector (in eSpace) is stored in the eInputVelocity local variable. You will see in a moment how this will be used if a satisfactory position cannot be found by the system in the requested number of slide iterations. When this is the case, we will call the detection phase one final time, passing in the initial velocity and simply return the position generated to the application. We need a backup of the initial velocity vector passed into the function in order to do this since the eVelocity vector will be continually overwritten in the response step with a new slide vector (to pass it back into the detection function in the next iteration).

In the above code, we use a helper function called Vec3VecScale which performs a per-component multiply between two vectors. Its code is a CCollision method and implemented inline as shown below.

```
inline static D3DXVECTOR3 Vec3VecScale (const D3DXVECTOR3&v1,const D3DXVECTOR3&v2 )
            { return D3DXVECTOR3( v1.x * v2.x, v1.y * v2.y, v1.z * v2.z ); }
```

In the next section of code we enter the actual collision detection and response phase. As discussed previously, there are times when the ellipsoid could get stuck in a sliding loop before the velocity vector is projected away to nothing. We control the maximum number of times the response phase will call the detection phase using a member variable called m_nMaxIterations. In virtually all cases, the new position of the ellipsoid will be found after only a few iterations (a few slides) at most. However, by putting this limit on the number of iterations the collision system will run, we can control the case where the ellipsoid is sliding repeatedly back and forth between two or more surfaces.

As you can see in the next code snippet, the maximum number of iterations is governed by a for loop. Inside this loop, the detection and response phases will be executed per iteration. If, during any iteration, the final position of the ellipsoid is determined (i.e., there is no remaining velocity vector), we can exit the loop and return this position back to the caller. If the loop finishes all iterations, it means a position could not be determined in the maximum number of iterations such that there was no remaining velocity. In this (rare) case an additional step will have to be taken that we will see at the end of the function. Let us have a look at the first section of the loop:

```
    // Keep testing until we hit our max iteration limit
    for ( i = 0; i < m_nMaxIterations; ++i )
    {
        // Break out if our velocity is too small
        if ( D3DXVec3Length( &eVelocity ) < 1e-5f ) break;

        // Attempt scene intersection
        // Note : We are working totally in ellipsoid space at this point
        if ( EllipsoidIntersectScene( eFrom, Radius, eVelocity,
                                      m_pIntersections, IntersectionCount) )
        {
```

Inside the loop we first test to see if the currently length of eVelocity is zero (with tolerance). If so, we do not have to move the ellipsoid any further and we can exit from the loop. We know at this point that the eTo local variable will contain the current position the entity should be moved to and can be returned to the caller. If this is the first iteration of the loop, then eVelocity will contain the initial velocity vector passed in by the application (in eSpace) and eTo will contain the desired destination position as calculated above (eFrom + eVelocity). However, if this is not the first iteration of the loop then eVelocity will contain the slide vector that was calculated in the last response step (performed at the bottom of this loop). If this is zero, then the response step projected the remaining velocity from a previous detection step onto the sliding plane and created a new eVelocity vector with such insignificant length that it can be ignored. In this case, we can still exit from the loop since the eTo local variable will contain the eSpace position of the ellipsoid returned from the previous call to the detection function.

Next, in the above code, we feed the eSpace position and velocity into the collision detection function EllipsoidIntersectScene. Notice that we also pass in the radius vector of our ellipsoid which was passed in by the caller. Why do we need to do this if we are already passing in the position and velocity vectors in eSpace (where the ellipsoid is a bounding sphere)? Remember that the detection function will need to transform the vertices of each triangle it tests for intersection into eSpace also, and for that, it will need to divide the components of each vertex of each triangle by the radius of the ellipsoid. The detection function is also passed in an array of CollIntersect structures (this array is allocated in the CCollision constructor to a maximum size of 100) which on function return will contain the intersection information for the closest collision between the unit sphere and the environment (in eSpace). The variable passed by reference as the final parameter will contain the number of triangles that have been simultaneously intersected and thus, the number of elements in the m_pIntersections array that contain meaningful information.

The next few sections of code demonstrate what happens when the EllipsoidIntersectScene function returns true (i.e., an intersection has been determined at some point along the velocity vector). This means the response phase needs to be executed. Since we talked about the response phase as being a

separate component of the collision system, you might have assumed it would be a large and complex function. But in fact, the code executed in this next block comprises the entire response phase of the system we have discussed thus far.

```
            // Retrieve the first collision intersections
            CollIntersect & FirstIntersect = m_pIntersections[0];

            // Set the sphere position to the collision position
            // for the next iteration of testing
            eFrom = FirstIntersect.NewCenter;

            // Project the end of the velocity vector onto the collision plane
            // (at the sphere centre)
            fDistance = D3DXVec3Dot( &( eTo - FirstIntersect.NewCenter ),
                                     &FirstIntersect.IntersectNormal );

            eTo -= FirstIntersect.IntersectNormal * fDistance;
```

In the above code we alias the first element in the returned m_pIntersections array with the local variable FirstIntersect for ease of access. Although multiple elements may exist in this array containing the information for several triangles that may have been intersected, all of these triangles will have been intersected at the same time along the velocity vector and as such, each element in this array will contain the same eSpace sphere position in its NewCenter member. Remember, this is the furthest position the detection phase could theoretically move the sphere along the velocity vector before it intersects one or more triangles. Since we are only interested in getting the new sphere position, we can just extract this information from the first element in the array.

Notice how we copy the new position of the sphere into the eFrom vector, which previously described the position we wish to move our sphere from prior to the detection function being called. However, because the response step is about to be executed, this position is going to be the new 'from' position when the detection function is called again in the next iteration of the loop. The velocity vector eVelocity will also be overwritten with the new slide vector that will be generated. Therefore, it is as if we are saying, "Let us now run the collision detection function again, starting from this new position. We will test to see if we can slide along the new velocity vector (the slide vector) the next time an iteration of this loop is executed."

In the next line of code (shown above) we project any remaining velocity onto the sliding plane. eTo currently describes the position we wanted to reach before the detection function was called. The NewCenter member of the first CollIntersect structure contains the position we are able to move to along the velocity vector. If we subtract the new position vector from the intended destination, we get the remaining velocity vector for the sphere. If we perform a dot product between this vector and the slide plane normal (returned from the detection phase also), we get the distance from the original eTo position (Original Dest in Figure 12.14) that we wanted to travel to, to the center of the sphere along the plane normal (D in Figure 12.14). That is, if the remaining velocity vector forms the hypotenuse of a right angled triangle and the new position of the sphere describes the point at which the hypotenuse and the opposite side of the triangle meet, dotting the vector **eTo - NewCenter** (Original Dest – Modified Dest in Figure 12.14) with the intersection plane normal, describes the length of the adjacent side of the triangle. As you can see in the final line of the code shown above, once we have calculated this distance, we can simply move eTo back along the plane normal (negative direction) so that it describes the

position where the opposite side meets the adjacent side (the blue arrow in Figure 12.14). This is a vector that is situated in the direction tangent to the slide plane that our sphere should slide along. Therefore, if we subtract this new eTo position from the new position of our sphere returned by the detection function, we have a new velocity vector describing the direction and magnitude we wish to slide along the plane when the collision detection phase is called in the next iteration of the loop.

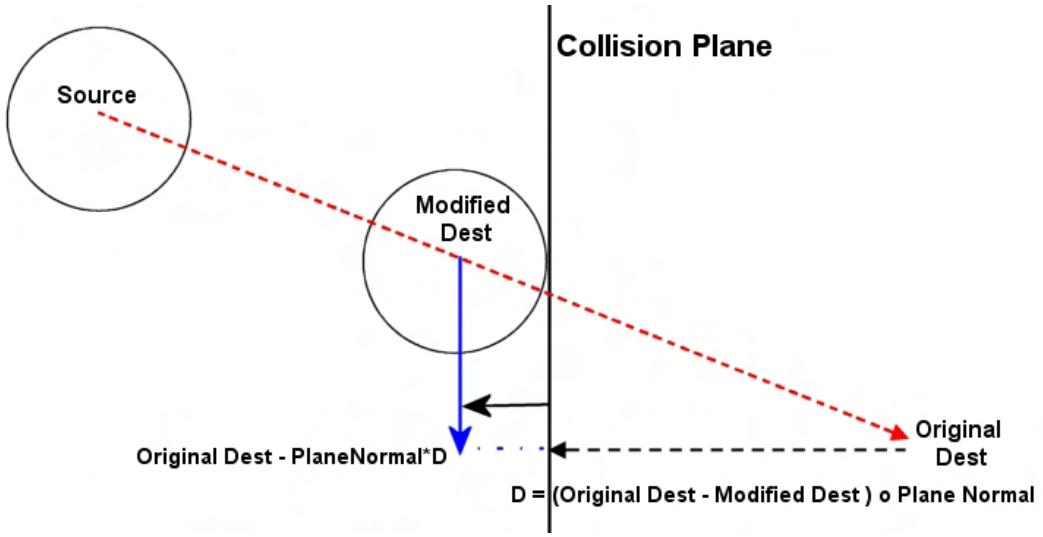


Figure 12.14

In the next section of code you will see how we calculate the eVelocity vector for the next iteration of the detection phase by doing what we just described -- subtracting the new position of the sphere from the projection of the remaining velocity vector onto the slide plane.

```
             // Update the velocity value
            eVelocity = eTo - eFrom;

            // Calculate new integration velocity
            fDistance = D3DXVec3Dot( &vecOutputVelocity,
                                     &FirstIntersect.IntersectNormal );
            vecOutputVelocity -= FirstIntersect.IntersectNormal * fDistance;

            // We hit something
            bHit = true;
```

In the above code we also project the current integration velocity (which in the first iteration will just be the input velocity) onto the slide plane. Notice that this is done in almost an identical way to the previous projection except that we project the whole velocity vector and not just the remaining portion behind the point of intersection. That is, we imagine that vecOutputVelocity describes a point relative to the origin of the coordinate system, and the slide plane normal passes through the origin. The dot product will produce fDistance which is the distance from that point to the plane along the plane normal. We then move the point (vecOutputVelocity) onto the plane by moving it towards the direction of the plane for a distance that places that point on the plane. This is our new integration velocity unless it gets changed in the next iteration. As you can see, this vector is continually projected onto the new slide plane in its entirety to preserve as much of the original velocity vector's length as possible.

Notice that in the above code we also set the local Boolean variable bHit to true so that we can tell, at the bottom of the function, whether at least one collision occurred. We will see how this is used in a moment.

Take a moment to study what we have done above. To reiterate, we have recalculated the eVelocity vector to store the slide vector, the eFrom vector now contains the new modified position of the sphere butted up against the colliding polygon. We have also updated the eTo vector describing the position we wish to travel to in the next iteration of the loop. We have essentially overwritten all the inputs that will be fed into the next iteration of the detection function. As far as the detection function is concerned, in the next iteration, these values could have been passed in by the application. They have of course been calculated by the response step to force a new movement into another location along the slide vector. The integration velocity is calculated using the same projection, but the entire vector is projected, not just the remaining velocity. Remember, this is the new velocity vector that will be returned to the caller, so we do not want this to be whittled away to zero.

We have essentially just covered the entire response step. There is just one additional line we put in to help reduce jittering that might occur when the player is trying to force the entity into an internal corner as shown in Figure 12.15.

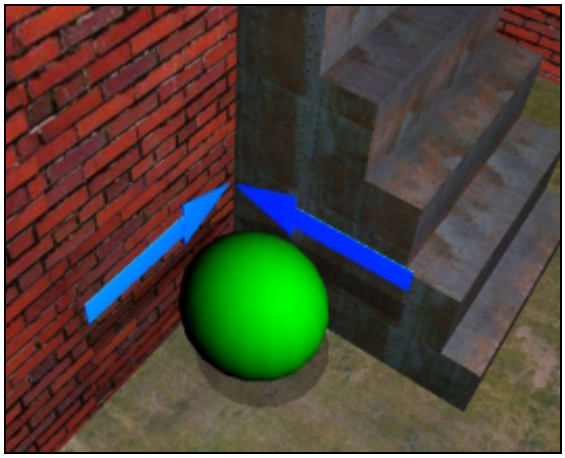


Figure 12.15

If the entity is forced into an internal corner or lip where polygons meet at an angle of 90 degrees or less (see Figure 12.15) a vibration artifact can occur until the initial velocity vector is completely spent. We might imagine that the sphere in Figure 12.15 first hits the wall in the corner which the response step slides into the corner and then into the side of the steps. When the sphere intersects with the side of the steps in the next detection step, it gets slid back into the wall it just slid along causing it to slide back in the opposite direction. When the sphere hits the wall again in the next detection phase, the response phase once again generates the original slide vector pushing it back into the side of the steps again, and so on. The sphere would seem to bounce between the two faces that form the corner in a cyclical pattern producing an unattractive vibration. To help resolve this, the response step can test if the slide vector (now stored in eVelocity ready for the next iteration) is pointing in the opposite direction to the original velocity vector that was passed in. That is, we can detect whether we are generating a slide vector that forms an angle greater than 90 degrees with respect to the original velocity vector that was first input by the application. If so, then we know that we are trying to move the sphere in a direction opposite of what was originally intended and have successfully detected where such a pattern of bouncing might be about to be carried out. When this is the case, we can simply use the current sphere position (returned from the last call of the detection function) as the final resting place of the sphere and exit the loop. The final line of code for the response step is shown below and performs this test.

```
            // Filter 'Impulse' jumps
            if ( D3DXVec3Dot( &eVelocity, &eInputVelocity ) < 0 )
                { eTo = eFrom; break; }
```

```
        } // End if we got some intersections
        else
        {
            // We found no collisions, so break out of the loop
            break;
        } // End if no collision

    } // Next Iteration
```

Studying the above code you can now see why we originally made a backup of the original velocity vector in eSpace (eInputVelocity) at the start of the function -- so we can filter out any vibration artifacts. And that is basically all there is to our main collision detection and response loop.

In the above code we also show the ‘else’ code block. This is executed when no collisions between the velocity vector and the scene are detected (i.e., EllipsoidIntersectScene returns false). In such a case, we break from the loop immediately since we already have the final resting place of the sphere in eSpace in the eTo variable. Either way, eTo will always contain the final resting place of the sphere in eSpace when this loop is exited. Notice that when we detect a looping pattern in the bottom of the response step in the above code, we copy the current position stored in eFrom to the eTo variable. Remember that this variable contains the new sphere position that was returned from the previous call to the detection function. Since we have discovered a looping pattern, we will decide to stop doing any further response processing and set this as the final resting place of the sphere.

At this point we have exited the loop and one of two conditions will be true; we will have exited the loop prematurely (the usual case) and we have the final resting position of the sphere in eSpace stored in the eTo variable, or the loop will have exited naturally after its full number of iterations. The second case is bad news since it means a very rare case has arisen where we were unable to spend the entire velocity in the maximum number of iterations to resolve a final resting place for the sphere. As discussed earlier, this can happen very infrequently when the geometry forms an angle that causes the sphere to repeatedly slide between opposing surfaces without significantly diminishing the velocity vector each time it is projected onto the sliding plane. If the maximum number of iterations was executed, then we have decided we no longer wish to spend any more processing time sliding this sphere around.

In the first case, we have our final position in eTo, so all we have to do is transform it back into world space. Just as we transform a position vector into eSpace by dividing its components by the radius vector of the ellipse, we transform an eSpace vector back into world space by multiplying its vector components by the radius vector of the ellipse (see below). This code is only executed if the loop variable is smaller than the maximum number of iterations (i.e., a final resting place for the sphere was resolved in the given number of loops and the loop exited prematurely). The final world space position for the ellipsoid is stored in the local variable vecOutputPos.

```
    // Did we register any intersection at all?
    if ( bHit )
    {
        // Did we finish neatly or not?
        if ( i < m_nMaxIterations )
        {
            // Return our final position in world space
```

```
            vecOutputPos = Vec3VecScale( eTo, Radius );

        } // End if in clear space
```

When this is not the case and we have executed the maximum number of iterations of the loop, we have to 'hack' around the problem. We will simply ignore everything we have done thus far and simply call the collision detection routine one more time with the original eSpace values. We will then take the position returned by the detection function (first point of impact along the original velocity vector) and return that as the final position of the sphere.

```
        else
        {
            // Just find the closest intersection
            eFrom = Vec3VecScale( Center, InvRadius );

            // Attempt to intersect the scene
            IntersectionCount = 0;
            EllipsoidIntersectScene( eFrom, Radius, eInputVelocity,
                                     m_pIntersections, IntersectionCount )
            vecOutputPos = Vec3VecScale( m_pIntersections[0].NewCenter, Radius );

        } // End if bad situation
    } // End if intersection found
```

As you can see in the above code, we reset the eFrom vector to be the eSpace version of the original position vector passed into the function and run the detection routine again. We then transform the intersection position into world space before storing it in vecOutputPos.

Our job is now done. Whatever the outcome, we now have the position the ellipsoid should be moved to stored in the local variable vecOutputPos and the new integration velocity vector stored in vecOutputVelocity. As a final step we will copy these values into the NewCenter and IntegrationVelocity variables which were passed by reference by the caller. The application will now have access to this new position and velocity and can use it to update the position of the entity which the ellipsoid is approximating. We then return either 'true' or 'false' to indicate whether any intersection and path correction took place.

```
    // Store the resulting output values
    NewCenter            = vecOutputPos;
    IntegrationVelocity  = vecOutputVelocity;

    // Return hit code
    return bHit;
}
```

And there we have it. With the exception of our examination of the EllipsoidIntersectScene method which actually calculates the new non-intersecting position of the unit sphere, we have walked through the entire collision system. Hopefully you will find this to be relatively straightforward, especially the response step which, while very effective, was actually quite small.

> **Note**: If you are comparing the code described above with the version in Lab Project 12.1's source code, you will likely notice that the version in the source code looks much more complicated and quite different in places. This is because the version supplied with Lab Project 12.1 has had a lot of extras added, such as the ability to provide collision detection and response against moving meshes in the environment (lifts, automated doors, etc). At the moment, we are just concentrating on the core system that deals with a static collision environment. As we progress through this chapter and the more advanced features are added to our collision system, you will see the code snippets start to look more like those in the accompanying lab project.

Of course, there is still a large portion of the collision system that requires explanation. We have not yet discussed how the EllipsoidIntersectScene function determines intersections with the sphere along its velocity vector and how it returns the new position of the sphere at the point of intersection. Moving forward, we will begin to focus on the implementation of the collision detection phase (i.e., EllipsoidIntersectScene and its helper functions).

12.4 Collision Detection

Our collision detection phase will be responsible for testing the movement of a sphere along a velocity vector and determining the distance that the sphere can travel along that vector before an intersection occurs between the surface of the sphere and the environment. The collision detection phase will comprise several intersection tests between rays and various geometric primitive types. While it might not be obvious why this is the case at the moment, we will need to cover how to intersect a ray with a triangle, a cylinder, and a sphere. These intersections tests will be performed one after another to determine whether the path is clear. Only when all intersection tests fail for every potential collider do we know that the velocity vector is clear to move the sphere along.

Also, several of the intersection tests may return different collision times along the velocity vector for a given triangle. This is because each uses a different technique to test the different parts of a triangle with the sphere. The edges, the vertices, and the interior of each triangle will all need to be tested separately to make sure they do not intersect the sphere at some point along its velocity vector. If several of the tests return different intersection positions along the velocity vector for a single triangle, our collision system will only be interesting in recording the intersection information with the smallest *t* value (the first collision along the path). In other words, we are only interesting in finding the furthest point the center of the sphere can move to along the velocity vector until its first impact with the environment occurs.

The detection process is complex because our sphere is (theoretically) moving along the velocity vector. Simple tests that could be performed with a static sphere cannot be used here because our entity is assumed to be in motion. We cannot perform a sphere/triangle test at both the start and end points of the velocity vector since that would ignore collisions that might have happened between the endpoints. We can easily imagine a situation where a sphere starts off in front of a triangle and is not intersecting it. Assume that we wish to move to a position (given a velocity vector) completely behind the triangle, which we know is also non-intersecting. Intuitively we see that in order for the sphere to get from one side of the triangle to the other, it must pass right through the triangle and thus must collide with it at some point along that path. Simply testing the start and end positions of the sphere would prove negative

for intersection, which might lead us to believe the sphere can indeed be moved across the full extent of the velocity vector to its desired destination.

Solving this problem requires that we account for the time aspect of the velocity vector. We will use a technique referred to as swept sphere collision detection, which accounts for the full travel path, including endpoints. A swept sphere is shown in Figure 12.16 and as you can see, it is like we have stretched (i.e., swept) the sphere along its velocity vector to create a capsule shape. Source describes the initial center position of the sphere and Dest describes the desired destination for the center of the sphere (Center + Velocity). In between, we see a solid area representing the integration of the sphere at each moment in time in which the sphere is traveling from origin to destination.

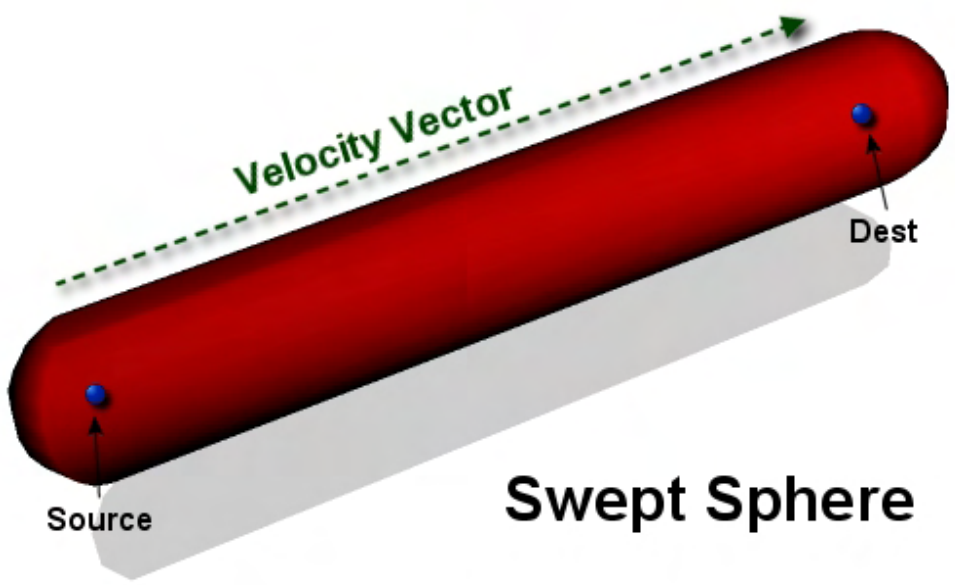


Figure 12.16

We know that if any geometry intersects the swept sphere then it must intersect the sphere at some point along the velocity vector. It is as if we have thickened the velocity vector ray by the radius of the sphere and are now testing this against the scene instead of the sphere itself. It covers all the space in the environment that the sphere will touch when moving along the velocity vector.

When we visualize the swept sphere as a primitive (as shown above) we can see that it bounds all the space that will be touched by the sphere on its path along the velocity vector. While this is vaguely interesting we still have not covered how to intersect such a shape with the environment. As you will see in a moment, we can greatly simplify this process by performing a transformation on any triangle that is to be tested against the swept sphere. Instead of creating a capsule shape as shown in Figure 12.6 describing the velocity vector thickened by the sphere radius, we can approach this problem from the other angle. We will see later that if we instead ‘inflate’ the geometry of our environment by the radius of the sphere, we can treat the sphere’s movement along the velocity vector, not as a capsule, but as a simple ray. We can test for intersections between this ray and the ‘inflated’ environment to determine the point at which the sphere surface intersects the environment. This technique will be used for testing the sphere’s path against the center of the polygon, the polygon edges, and the polygon vertices. It should be clear then that our collision detection techniques will essentially be a collection of methods

that intersect a ray with other geometric primitive types. Therefore, before we try to understand the collision detection system as a whole, let us spend some time defining what a ray is and look at some of the various intersection techniques we can perform with rays. Once we understand how to intersect a ray with several different primitive types, we will then see how these intersection techniques can be assembled to create our core collision detection system.

12.4.1 Rays

In this section we will discuss rays and how they can be represented and used to perform intersection tests. It is important that we initially get our terminology right because in classical geometry there is a distinct difference between a line, a line segment, and a ray.

- A **Line** has no start and end point and is assumed to extend infinitely in both directions along a given slope.
- A **Line Segment** is a finite portion of a line in that it has a start and an end position in the world. To visualize a line segment we would draw a mental line between the two position vectors describing the start and end points. Thus, it is quite common for a line segment to be represented by two positional vectors.
- A **Ray** has a starting position (an origin) and a direction. It is assumed to extend infinitely in the direction it is pointing. We can think of a Ray as being one half of a line. Whereas a line would extend infinitely from the starting position of the ray in both the positive and negative directions, a Ray will start at some origin and extend infinitely in one direction only -- the direction the ray is pointing.

Figure 12.17 demonstrates the differences between these three geometric primitives.

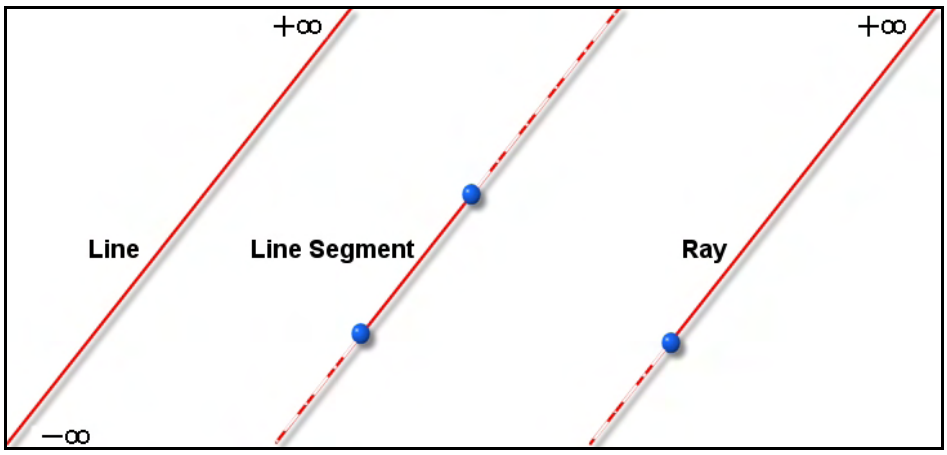


Figure 12.17 : A Line, A Line Segment and a Ray

In computer science, we will sometimes take some liberties with these definitions, and this is especially true in the case of a ray. While it is often useful to think of a ray has having a starting position and a direction in which the ray extends infinitely, it is also useful to use a ray definition that allows for a finite length from the origin. If we think about the velocity vector passed into our collision routines, we

can think of this as being a ray of finite length when coupled with the center position of the sphere. The sphere center point describes the origin of the ray and the velocity vector describes the direction of the ray and its magnitude. This is the ray representation we will be using most in this chapter and throughout our ray intersection routines. Since we can add the ray's direction vector to its origin to get the end point of the ray, we can see that our ray is really just a directed line segment. In other words, it is a line segment that is described not as two position vectors, but as a single position vector (the ray origin) and a vector describing the direction the ray is traveling from the origin and the distance it travels in that direction. Therefore, in our discussions we will use the following definition:

A **Ray** is a directed line segment.

Let us now take a look at how we will define our ray. A 3D ray is often specified in its parametric form as shown below using three functions.

Parametric Form of a Ray (3D)

$$x(t) = x_o + t\Delta x$$
$$y(t) = y_o + t\Delta y$$
$$z(t) = z_o + t\Delta z$$

where

$(x_o \;\; y_o \;\; z_o)$ = **Ray Origin (Start Position)**

$(\Delta x \;\; \Delta y \;\; \Delta z)$ = **Direction (Delta) Vector**

The parameter to each of these functions is *t*, where *t* = [0.0, 1.0]. The combined results of each function will form a vector that describes a position somewhere along the ray where *t* is between 0.0 and 1.0. Note that *t* is the same input to each function. Each function shown above is simply describing a movement along the direction vector (Δi) from the ray origin (i_o) for each component x, y, and z. The result of each component calculated can then be linearly combined into the final position vector (x, y, z) describing the point along the ray given by *t*. It is often nicer to represent a 3D ray parametrically using its more compact vector form:

$$p(t) = p_o + td$$

where

p_o = A 3D vector describing the position of the start of the ray (the origin)

d = A non-unit length 3D vector (or a unit length vector if you wish the length of the ray to be 1.0) describing the direction and length of the ray.

We can describe any position on that ray by specifying a function input value of *t* between 0.0 and 1.0. For example, imagine that the ray origin **p** is a vector describing a ray origin <100, 100, 100>. Also imagine that the direction and magnitude of the ray described in vector **d** sets the end of the ray at a position offset 10 units along the X, Y and Z axes from the ray origin. That is, <10, 10, 10>. To calculate a position exactly halfway along that ray we can use a *t* value of 0.5, as shown below.

Position = (100, 100, 100) + t (10, 10, 10)
= (100, 100, 100) + 0.5 (10, 10, 10)
= (100, 100, 100) + (5, 5, 5)
= (105, 105, 105)

Using a *t* parameter of 0.5, we have correctly calculated that the exact position of a point situated exactly halfway along the ray is (105, 105, 105).

In Figure 12.8 we see the same ray and the calculation of other points along the ray. We can see for example that a *t* value of 0.25 gives us a position situated ¼ of the way along the ray from the ray origin. We can also see that using a *t* value of 0.75 provides us with a position situated exactly ¾ of the length of the ray from the origin. Finally, we can see that the end point of the ray can be calculated using a *t* value of 1.0. As the function for calculating the end position (*t* = 1) will simply scale the ray direction vector by 1.0, we can omit the *t* value altogether when we wish to calculate the end of the ray by simply adding the direction vector to the ray origin:

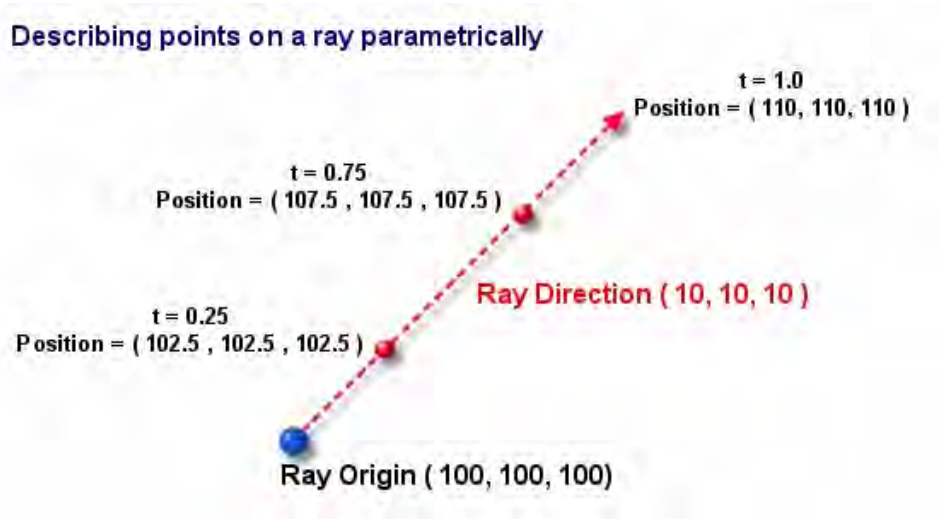


Figure 12.18

Position = (100, 100, 100) + 1.0 (10, 10, 10)
= (100, 100, 100) + (10, 10, 10)
= (110, 110, 110)

Alternatively, we can think of *t* as describing how many units we would like to move along the ray measured in units of 'ray length'. A value of 1.0 means we would like to travel one whole ray length and a value of 0.5 would describe us as wanting to move along the ray ½ of the ray length. Even when representing a ray in its classical geometry form, this same relationship holds true, although it may not be immediately obvious. The classical geometry representation of a ray is that it has a position and a

direction and an infinite length. Such a ray definition can be represented in the same parametric vector form we have been using thus far:

$$p(t) = p_o + td$$

The only difference between this representation and the one we have been using is that it is implied that ***d*** is a unit length vector. It describes a direction only, but no magnitude since the length of the ray is infinite (controlled by *t* which in this case has an infinite range). When using such a definition we will usually have some other variable that determines legal values for points along the ray. For example, we might have a variable *k* that contains the length of the ray. Positional vectors on the ray that have a distance from the origin that is larger than *k* are considered to be beyond the legal end point of the ray.

When using the classical representation of a ray, the *t* value still essentially means the same thing -- it describes the number of units (in vector length) we would like to move along the ray from the origin. However, with the classical definition, the *t* value will not be restricted between 0.0 and 1.0, but rather between 0.0 and *k*, where *k* is some arbitrary value used to describe the length of the ray. Using the classical definition we would use a *t* value to describe a position exactly *t* units along the infinite ray. A value of $t = 7$ for example would describe a point along the ray seven units away from the origin, when in the case of the classical definition, the units of measurement are the units of the coordinate system for which it is defined (e.g., world space). Obviously, this seems quite different from the ray definition we have been using thus far, where units are based on 'ray length'. However, if we think about the classical ray representation where a unit length direction vector is used, we could hold the view that instead of the ray being infinite, it has a length of 1 (because the direction vector ***d*** is unit length). In this case, we can see that a value of $t = 7$ does indeed still describe a position along the ray measured in units of 'ray length'. In the classical case, ray length is 1.0, so the mapping between the ray length and the units of the coordinate system for which it is defined is 1:1.

Although we will be using the definition of a ray which involves a delta vector, that does not mean that we cannot create a ray in this form from information stored in another form. Very often we may not have the delta vector at hand and may have only the start and end points that we would like the build the ray between (i.e. the line segment). That is not a problem since we can create the delta value by subtracting the start position from the end position. The resulting vector ***d*** will describe the direction from the first vector to the second and will have the length of that ray encoded as its magnitude. The example below shows how easy it is to calculate the delta vector for our ray when the initial information is in the form of two positional vectors:

RayStart = (100, 100, 100)
RayEnd = (110, 110, 100)
d = RayEnd – RayStart = (10, 10, 10)

Exactly the same technique can be used to create the unit length direction vector for the classical geometry representation of the ray. An additional step is required to normalize vector ***d***. The following example shows how we could create a ray in classical form and then use an additional variable *k* to describe the length of the ray. Any *t* value larger than *k* would be considered past the end of the ray.

RayStart = (100, 100, 100)
RayEnd = (110, 110, 100)

d = RayEnd - RayStart = (10, 10, 10)
d = Normalized (d)

k = $\sqrt{10^2 + 10^2 + 10^2}$ = **17.3205**

We now know what a ray is and how we can represent one using two vectors (a ray origin and a direction). Because we will be using the non-classical geometry definition of a ray, our direction vector will also have magnitude and will be referred to as a delta vector. When using delta vectors, points on the ray can be described parametrically using *t* values between 0.0 and 1.0. Any *t* value that is smaller than zero is said to exist on the same line as the ray, but exists before the ray origin. Any *t* values that are larger than 1.0 describe points that exist on the same line as the ray but after the ray end point.

With the description of a ray behind us, let us now move on and look at some intersection tests that can be performed using rays. This will help us uncover why rays are so useful in 3D computer graphics (and in particular, in our collision detection system).

12.4.2 Ray / Plane Intersection Testing

It only takes a small amount of experience to realize that the dot product is the veritable Swiss Army Knife of 3D graphics programming. It is used to perform so many common tasks in 3D graphics programming that it would be impossible to make a 3D game without it. The dot product comes to the rescue again when trying to determine whether a ray intersects a plane. The process is remarkably simple as long as you understand how the dot product operation works.

We have discussed several times in the past (see Graphics Programming Module I) the properties of the dot product and we even examined ray intersections with a plane in our workbooks. We approached the understanding of this technique by analyzing the geometry and applying the properties of the dot product to solve the problem. But this method is not always ideal when handling more complex intersections since drawing diagrams to show geometric solutions can become a very ungainly process.

It is generally much simpler to come up with a pure mathematical solution. That is, we can find solutions by substituting the ray equation into the equation for the primitive we are testing against and then perform some algebraic manipulation to solve for *t* (the interval along the ray where the ray intersects the primitive). This is a technique we managed to steer clear of in the past because other solutions were available. We did not have to worry about your ability to solve equations using algebra. But we can put this off no longer. So we will start with the ray/plane intersection technique as a refresher for the math we (hopefully) learned in high school. We will start with the more familiar geometry perspective and then will solve the same problem algebraically to get us warmed up for what is to come later in the chapter. Do not fear if your algebra skills are a little rusty. We will review everything in detail and you will quickly see just how easy it can be once you get the hang of it again.

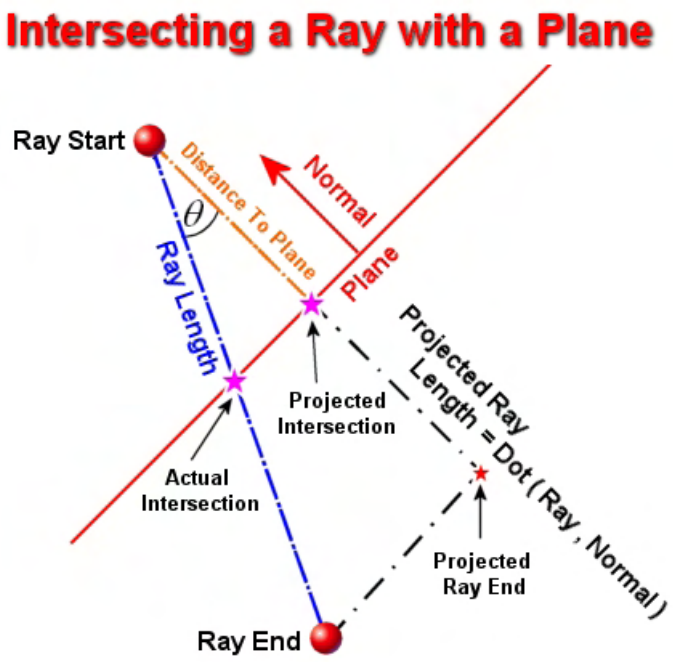


Figure 12.19

Study the diagram in Figure 12.19. Ray Start describes the position of the ray origin and the blue dashed line describes the delta vector. We can see that if we add the delta vector to the ray origin we would get the ray end position. We wish to find the position along the ray where intersection with the plane occurs. This can be done quite simply by applying the properties of the dot product and projecting the ray delta vector along the plane normal.

Our first task is to calculate the distance from the ray origin to the plane. Recall that the plane equation is:

$$ax + by + cz + d = 0$$

In the plane equation **a**, **b,** and **c** are the x, y and z components of the plane normal and **d** is the signed distance from the origin of the coordinate system to the plane along the plane normal. The **x**, **y**, and **z** variables in the plane equation describe the 3D components of the point on the plane. For the equation to be satisfied, the point (x,y,z) must lay on the plane described by (a,b,c,d). As both the position of the point and the plane normal are 3D vectors, we can write the plane equation as a dot product between the point and the plane normal with the addition of the plane distance. The plane equation in its more compact vector form looks like this:

$$P \bullet N + d = 0$$

where **P** is the point on the plane and **N** is the plane normal. It should also be noted that you could store the plane coefficients in a 4D vector and represent the point on the plane as an homogeneous 4D coordinate with w=1. The classification of the point against the plane is then simplified to a 4D dot product. As can be seen, the plane equation will return the distance from the point to the plane (which will be zero for any point on the plane). Therefore, if the plane equation is satisfied, the point **P** is located on plane. For any point not on the plane, the result will be the signed distance to the plane.

While it is most common for a plane to be stored in **a**, **b**, **c**, **d** format, it is also fairly common to find a plane represented using the plane normal and a point known to be on the plane. When this is the case, the **d** plane coefficient can be calculated by taking the negative dot product of the plane normal and the point on the plane. This can be seen by looking at the plane equation. To solve for **d** we have to leave **d** on its own on the left hand side of the equation. We can do this by subtracting $P \bullet N$ from the left and right side of the equation. Remember that anything we do to one side of the equation must be done to the other, so that the equation remains balanced. So we start with:

$$(P \bullet N) + d = 0$$

If P is our point on the plane, then we wish to subtract $P \bullet N$ from both sides of the equation. Subtracting it from the LHS (left hand side of the equals sign) leaves us with just **d** and we have successfully isolated **d**. We then have to perform the same operation to the RHS of the equation to keep it balanced. Subtracting $P \bullet N$ from zero gives us:

$$d = -(P \bullet N)$$

This can also be written as:

$$d = -N \bullet P$$

This is commonly how you will see **d** calculated.

Sometimes people will use the plane equation in the form:

$$ax + bx + cz - d = 0$$

In this case the vector form is obviously:

$$(P \bullet N) - d = 0$$

When using this form of the equation to solve for a distance, the **d** coefficient must be calculated slightly differently. Let us manipulate this form of the plane equation to see how **d** must be calculated. To isolate **d** we add **d** to both sides of the equation. This will cancel out **d** on the LHS and introduce it to the RHS.

$$P \bullet N = d$$

or in the more familiar form:

$$d = P \bullet N$$

So as you can see, if we intend to use the ax+by+cz+d version of the equation, d must be calculated by negating the result of the dot product between the plane normal and the point on the plane. If we intend to use the ax+by+cz-d version of the equation, d must be calculated by taking the dot product of the plane normal and the point on plane and not negating the result. This subtle difference in the calculation of d is what makes both forms of the equation essentially the same. Please refer back to Module I in this series for more detailed discussion on the positive and negative versions of the equation if you need more in depth coverage.

Getting back to our problem, we first must calculate the distance from the ray origin (Ray Start) to the plane. We do this as follows:

PlaneDist = $RayStart \bullet Normal + D$

This gives us the distance from the ray origin to the plane along the direction of the plane normal. This is shown as the orange dashed line in Figure 12.19 labelled Distance To Plane. We can also calculate the distance to the plane from the origin in another way. We can simply dot the plane normal with a vector formed from a point known to be on the plane to the ray origin. You will see us using this technique many times in our code.

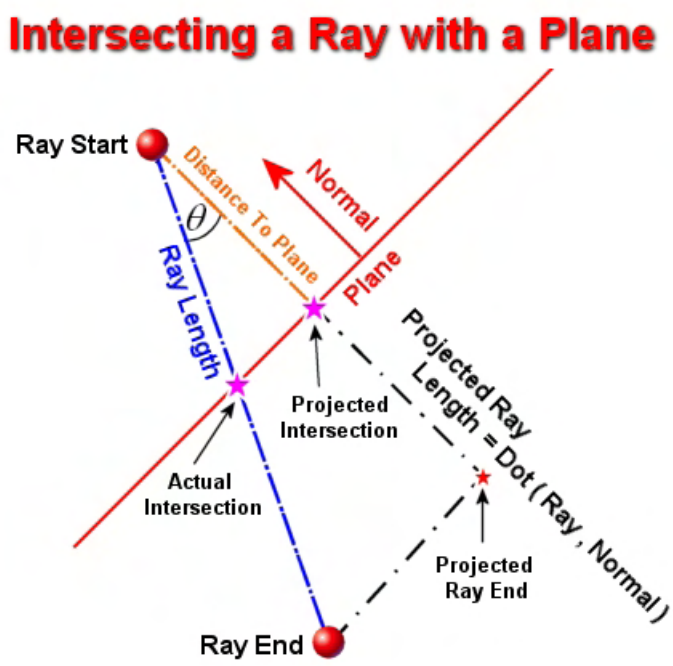


Figure 12.19 (again for convenience)

Our next task is to make sure we have the ray delta vector. Our intersection routines will use the ray in the form of an origin and a delta vector, so if the only information at hand is the start and end positions (as can sometimes be the case) we can calculate the delta vector by subtracting the ray start position from the ray end position. If we imagine that the line labelled Distance To Plane is our ray (instead of the actual ray) we would instantly know the distance to the intersection -- it is simply the distance to the plane from the start point which we have already calculated. However, our ray is travelling in an arbitrary direction and we need to know the time of intersection along this ray, not along the direction of the plane normal. But if we take the dot product of the negated plane normal and the ray delta vector ***d*** to get them both facing the same way (see the blue dashed line in Figure 12.19), we will get the length of the delta vector projected onto the plane normal. This describes the length of the ray delta vector as if it had been rotated around the ray origin and aligned with the plane normal.

Projected Line Length = $d \bullet -Normal$

Remember that the plane normal is a unit length vector but the ray delta vector is not. The result of a dot product between a unit length vector and a non unit length vector is:

$$\cos(\theta)|Ray| = d \bullet -Normal$$

In other words, the result is the length of the non unit length vector (the ray delta vector) scaled by the cosine of the angle between them. If you look at the image, we can see that this would give us the length of a vector from the ray origin to the Projected Ray End point along the direction of the negated plane normal. If we imagine the ray delta vector to be the hypotenuse of a right angled triangle, the result describes the length by which we would need to scale the unit length normal vector such that it correctly described the adjacent side of the same right angled triangle. If we were to scale the negated normal of the plane by this result, we would get the projected ray end point shown in the diagram. If we were to

draw a straight line from the actual ray end point to the projected ray end point we can see that this would form the opposite side of a right angled triangle also. So, by dotting the ray delta vector with the negated normal, we end up with the length of the ray projected along the plane normal.

So what does this tell us? Quite a lot, actually. With the projected ray length, we can measure the intersection between the projected ray and the plane much more easily. We know from looking at the diagram that the point at which the projected ray intersects the plane in this new projected direction is simply the distance to the plane from the ray start point. This is the orange line in the diagram labelled Distance to Plane, which we have already calculated.

We know the length of the projected ray and the distance at which the plane intersects the projected ray because now they both share the same orientation. The intersection point is simply the distance to the plane from the ray origin. If we divide the distance to the plane (the distance to the intersection along the projected ray direction) by the length of the projected ray, we get back a float *t* that represents the intersection along the projected ray as a percentage between 0.0 and 1.0. It essentially describes the distance to the intersection in units of overall ray length. So a *t* value of 0.5 would mean the intersection is exactly halfway along the ray's delta vector.

You may see versions of the calculation of the *t* value that negate the sign of the Distance to Plane when performing such a calculation. It depends on whether you calculate your distance to plane value as a positive or negative value when a point is in front of the plane. You may also see versions of this calculation where the Projected Ray Length is negated before the divide. This boils down to whether it was projected by performing $d \bullet -Normal$ as we have, or by using $d \bullet Normal$. In the second case, the dot product will return a negative value for the projected line length because the delta vector ***d*** and the non-negated plane normal are facing in opposite directions. All that is happening is that the negation is not being applied to the normal; it is being applied later to the result. However, for the time being, we are assuming that we have calculated the distance to the plane and the projected ray length as positive values by flipping the plane normals where applicable.

t = Distance To Plane / Projected Ray Length

If Projected Ray length ($d \bullet -Normal$) equals zero then it means no intersection occurs because there is a 90 degree angle formed between the plane normal and the ray delta vector. This means the ray is parallel with the plane and cannot possibly intersect it. If the distance to the plane from the ray origin is a negative value, it means the ray origin is already behind the plane. You may or may not want to consider intersections with the back of a plane depending on your application's needs, so you may or may not wish to return false immediately when this is the case. If $t < 0$ then the ray origin is situated behind the plane and will never intersect the back of it. If $t > 1$ then the intersection with the plane happens past the end point of the ray and is not considered to be a valid intersection. A *t* value in the range of 0.0 to 1.0 describes the intersection with the plane along the projected ray and the intersection along the actual ray parametrically. Thus, if we wanted our Ray/Plane intersection method to return the position of intersection (and not just the *t* value) we could easily calculate it as follows:

Intersection Point = Ray Origin + (Ray Delta * t)

As you can see, we are just scaling the ray delta vector by the *t* value to reduce the delta vector's length such that it correctly describes the length of a vector from the ray origin to the intersection with the plane. We then add this to the ray origin to get the actual position of the intersection with the plane.

Taking what we have just learned, we can now implement a function that will calculate ray/plane intersections. It will take a ray origin, a ray delta vector, a plane normal, and a point on that plane as input parameters. We also have two output parameters: a 3D vector which will store the intersection point (if one occurs) and a float to store the *t* value of intersection.

```
bool RayIntersectPlane( const D3DXVECTOR3& Origin,
                        const D3DXVECTOR3& Direction,
                        const D3DXVECTOR3& PlaneNormal,
                        const D3DXVECTOR3& PlanePoint,
                        float& t,
                        D3DXVECTOR3& IntersectionPoint )
{
    // Calculate Distance To plane
    float PlaneDistance = D3DXVec3Dot( &(Origin - PlanePoint), &PlaneNormal );

    // Calculate Projected Ray Length
    float ProjRayLength = D3DXVec3Dot( &Direction, &-PlaneNormal );

    // If smaller then zero then it is either running parallel to the plane
    // or intersects the back of the plane. We choose to ignore in this version.
    if ( ProjRayLength < 1e-5f ) return false;

    // Calculate t
    t = PlaneDistance / ProjRayLength;

    // Plane is either before the start of the ray or past the end of the ray so
    // not intersection
    if ( t < 0.0f || t > 1.0f ) return false;

    // Calculate the intersection point
    IntersectionPoint = Origin + ( Direction * t );

    // We're intersecting
    return true;
}
```

In this version of the function we calculate the distance to the plane from the ray origin by projecting a vector from the point on the plane to the ray origin along the plane normal. This step would not be necessary if the function was passed all the plane coefficients and not just the normal.

Next we see another version of the function that does the same thing, only this time, the distance to the plane from the ray origin can be calculated using ax+by+cz+d. In this version, we pass the plane in as a single parameter (instead of a normal and a point) stored inside a D3DXPLANE structure. This is a structure with four float values labeled a, b, c, and d as expected.

```
bool RayIntersectPlane( const D3DXVECTOR3& Origin,
                        const D3DXVECTOR3& Direction,
                        const D3DXPLANE& Plane,
```

```
                          float& t,
                          D3DXVECTOR3& IntersectionPoint )
{
    // The D3DXPlaneDotCoord function performs the plane equation calculation.
    // It could be written manually as :
    // Plane Distance = Plane.a * Origin.x + Plane.b * Origin.y +
    //                  Plane.c * Origin.Z + Plane.d
    float PlaneDistance = D3DXPlaneDotCoord( &Plane , &Origin );

    // Calculate Projected Ray Length using the D3DXPlaneDotVector function
    // It performs ax + by + cz. A simple dot product between the plane normal
    // and the passed vector
    float ProjRayLength = D3DXPlaneDotVector( &Plane, &Direction );

    // if smaller then zero then it is either running parallel to the plane
    // or intersects the back of the plane we choose to ignore in this version.
    if ( ProjRayLength > -1e-5f ) return false;

    // Sign must be negated because Plane Normal was not inverted
    // prior to dot product above
    t = - ( PlaneDistance / ProjRayLength );

    // Plane is either before the start of the ray or past the end of the ray so
    // not intersection
    if ( t < 0.0f || t > 1.0f ) return false;

    // Calculate the intersection point
    IntersectionPoint = Origin + ( Direction * t );

    // We're intersecting
    return true;
}
```

This version of the function will probably appeal more to the math purists. This will make some sense when we try to arrive at an algebraic solution in a moment. Here we are using the standard plane equation **ax + by + cz + d** to calculate the distance to the plane from the ray origin. This is performed inside the function D3DXPlaneDotCoord. It essentially performs a 4D dot product between the plane coefficients and a 3D vector with an assumed *w* coordinate of 1.0 (1**d*).

We then calculate the projected line length by doing a dot product between the plane normal and the ray delta vector. We use the D3DXPlaneDotVector function for this which performs a 3D dot product between the a, b, and c components of the plane (the normal) and the x, y and z components of the input vector. You will recall that in the previous version of the function we negated the plane normal before we performed this dot product so that both the ray delta vector and the plane normal were facing the same way. In this version we have not done this (i.e., negating the a, b, and c coefficients of the plane prior) so the projected line length returned will be a negative value. Dividing the plane distance by this value will also generate a negative *t* value, which is not desirable. Notice how we take care of this later in the function simply by negating the result the of the *t* value calculation. This nicely demonstrates what was said earlier about the *t* value calculation potentially needing to have its result negated depending on the sign of the inputs to its equation.

The Algebraic Solution

In the last section we analyzed ray / plane intersection from a geometric perspective and arrived at our final *t* value calculation. As we proceed through this chapter however, we will have to be ready to find algebraic solutions since the problems we will attempt to solve will become more challenging and less easily represented from a geometric perspective. In order to refresh ourselves in the subject of algebra, we will start by finding a solution for Ray / Plane intersection using algebraic manipulation.

When using the standard plane equation in the last section, we discovered that to determine the value of *t*, the calculation was:

$$t = -\frac{\text{Distance To Plane}}{\text{Projected Ray Length}}$$

Let us see if we come up with the same equation using an algebraic approach.

Our ray **R** is defined parametrically and has an origin vector **O** and a delta vector **V**. *t* values in the range of 0.0 to 1.0 represent points on the ray.

$$R(t) = O + tV$$

Our plane is represented using the standard form of the plane equation:

$$ax + by + cz + d = 0$$

In vector form this can be written as the dot product between a point **P** (x,y,z), a plane normal **N** (a,b,c) and the addition of the d coefficient (the distance to the plane from the origin of the coordinate system).

$$P \bullet N + d = 0$$

Substituting the Ray into the Plane Equation

In the vector form of the plane equation, **P** represents a point on the plane if the equation is true. However, we wish to find a point on the ray which intersects the plane. Of course, if the ray intersects the plane, the point where it intersects is obviously on the plane. Thus, we are going to want to find the *t* value for our ray which would produce a point which makes the plane equation true. So we substitute **P** in the plane equation with our ray definition and we get:

$$(O + tV) \bullet N + d = 0$$

As the term (**O** + *t***V**) is dotted by **N**, this is equivalent to dotting both vectors (**O** and *t***V**) contained inside the brackets by **N** individually (the dot product is distributive). This means we can instead write **O** dot **N** + *t***V** dot **N** allowing us to expand out the brackets:

$$O \bullet N + tV \bullet N + d = 0$$

In order to find the point on the ray which intersects the plane, we have to find *t*. If we solve for *t* in the above equation, then we are finding the point on the ray which returns a distance of zero (look at the = 0 on the RHS) from the plane.

Solving such equations is actually quite simple. We just have to figure out a way to move everything we do not want to know about over to the right hand side of the equals sign, leaving us with the variable we do want to know about isolated on the left hand side of the equals sign (or vice versa). At this point, we will have solved the equation and will know how to calculate *t*.

Looking at the above equation we can see that the first thing we can get rid of on the LHS is '+*d*'. If we subtract *d* from the LHS of the equation it will cancel it out. We must also make sure that anything we do to the LHS must be done to the RHS to keep the equation balanced. If we subtract *d* from both sides of the equation, our equation now looks like this :

$$O \bullet N + tV \bullet N = -d$$

We also see that *t* is multiplied by ($V \bullet N$) on the LHS. If we divide the LHS by $V \bullet N$ we can cancel that out from the LHS. Of course, we must also divide the RHS by $V \bullet N$ as well to keep the equation balanced. Performing this step gives us the following:

$$\frac{O \bullet N}{V \bullet N} + t = \frac{-d}{V \bullet N}$$

t is very nearly isolated on the LHS now. All that is happening to *t* now on the LHS is that it is having $O \bullet N$ added to it. Therefore, we can completely isolate *t* on the LHS by subtracting $O \bullet N$ from both sides of the equation. This leaves us with the final solution for *t* as shown below:

$$t = \frac{-d - O \bullet N}{V \bullet N}$$

Of course, this can also be written by placing the numerator in brackets and distributing a -1 multiplication:

$$t = \frac{-(O \bullet N + d)}{V \bullet N}$$

This is the equivalent of the more common form:

$$t = -\frac{O \bullet N + d}{V \bullet N}$$

And here we see the final solution for *t*. Note that is it exactly how the last version of the RayIntersectPlane function calculated its *t* value:

```
t = - ( PlaneDistance / ProjRayLength );
```

Looking at our final equation, we know that $O \bullet N + d$ calculates the distance to the plane from the ray origin **O**. We also know that $V \bullet N$ computes the length of the ray projected onto the normal. However, because the normal **N** and the delta vector **V** are pointing in opposite directions, this will produce a negative length value. When the positive plane distance is divided by a negative projected length, we will get back a negative *t* value. This is why we negate the final result. As you can see, this is exactly what we have in the equation for *t*.

The solution for *t* could also be written by negating $V \bullet N$ before the divide like so.

$$t = \frac{O \bullet N + d}{-V \bullet N}$$

These are all identical ways of calculating the final positive *t* value with a negative projected length. You may see us using any of the forms to solve for *t* in our code from time to time. They all are equivalent.

12.4.3 Ray / Polygon Intersection Testing

Determining whether or not a ray intersects a plane is very a useful technique. It is used in polygon clipping for example, where each edge of the polygon can be used as a ray which is then intersected with the clip plane. We will cover polygon clipping in the next chapter when we discuss spatial partitioning techniques and you will see this all taking place. Ray/Plane intersection testing is also used when determining whether or not a ray is intersecting a polygon. This is important because ray/polygon intersections will be used by our collision detection system and by a host of other techniques we will employ later in this training program.

We will see one application of Ray/Polygon intersection testing later in this course when writing a Lightmap Compiler tool. A Lightmap Compiler is an application that will calculate and store lighting information in texture maps which can later be applied to polygons to achieve excellent per-texel lighting results. In such an application, a ray is used to represent a beam of light between each light source in the scene and the texels of each polygon's texture. If the light source ray reaches the polygon currently being tested then the lighting information is added to the light map texture. If the ray is

intersected by other polygons in the environment on its way to the polygon currently having its lightmap computed, then the polygon is said to be at least partially in shadow with respect to that light source and the light source does not contribute lighting information to that particular texture element. We will look at how to build our Lightmap Compiler later in the course. In fact, Ray/Polygon intersection tests will be used in all manner of lighting techniques and not just light mapping.

Another application of Ray/Polygon intersection is a technique called picking. When writing an application such as GILES™, we want the user to be able to select a polygon in the scene using the mouse as an input device. To accomplish this, the screen coordinates of the mouse can be converted into world space coordinates and then used to create a ray that extends into the scene from the camera position. The polygon intersection producing the smallest *t* value is assumed to be the first polygon that is hit by the ray and is the polygon that is selected. This is pretty much the same technique that is used by simple ray tracing renderers as well. In a simple ray tracer, the near view plane is essentially carved up into a pixel grid and rays are cast from the eye point (located some distance behind the plane) through each pixel location into the scene to find the point of closest intersection. The lighting information at that point is then computed and stored in the image pixel.

As you are no doubt convinced by now, the need to be able intersect rays with polygons is of paramount importance in 3D computer graphics.

Testing whether a ray intersects a polygon is a two step process. First we can perform a ray/plane intersection test to see whether the ray intersects the infinite plane that the polygon is on. The ray/plane test is done first because testing for a ray/plane intersection is computationally inexpensive. If a ray does not intersect the plane that a polygon lies on, it cannot possibly intersect the polygon itself. Therefore, performing the ray/plane intersection test first provides an early out mechanism for many of the polygons that we need to test which are not going to be intersecting the ray.

If a polygon passes the initial ray/plane test, the next test looks to see if the ray intersects the actual polygon itself. Since planes are infinite, even if the ray does intersect the plane of the polygon, the intersection with the plane may have happened well outside the borders of the polygon. So our second test will need to determine if the plane intersection point returned from the ray/plane intersection test is within all the edges of the polygon. If this is the case, then the point is indeed inside the polygon and we will have successfully determined that the ray has intersected the polygon. Since polygons can take on arbitrary shapes it might appear that a generic solution would be very hard to create. However, with our trusty dot product and cross product operations (see Module I), the task is mostly trivial.

Once a plane intersection point has been found, our job is to see whether the intersection point actually resides within the interior of the polygon. Since a polygon is just a subset of its parent plane, the testing process is really nothing more than finding out whether the intersection point is contained within all edges of the polygon. So how do we test this? Usually, your polygons will store their plane normals, but if this is not the case, you can refer back to Module I to see how to generate a polygon normal by performing the cross product on two of its edge vectors and normalizing the result. We will assume for the sake of this discussion that the polygon you wish to test for intersection also contains a polygon normal.

The “Point in Polygon” testing process is broken down into a set of simple tests (one for each edge of the polygon). For each edge we create an edge normal by performing a cross product between the edge vector and the polygon normal. For this test, the edge normal does not need to be normalized. This will return to us the normal of the edge and since we know that any vertex in the edge will suffice for a point that lay on the plane described by this edge normal, we have ourselves a complete description of the plane for that edge. We then simply classify the plane intersection point against the edge plane.

In the example shown in Figure 12.20, the edge planes are created with their normals facing outwards from the center of the polygon. Therefore, if the point is found to be in front of **any** edge plane, it is outside the polygon and we can return false immediately. If the point is behind the edge currently being tested, then we must continue to test the other edges. Only when the intersection point is contained behind all edge planes is the intersection point considered to be inside the polygon.

Figure 12.20 shows this process by demonstrating two example intersection points (P0 and P1) which are being tested for “Point in Poly” compliance. P0 is clearly inside the polygon because it lies behind all the planes formed by the edges of the polygon. P1 is not inside because it does not lay behind all edge planes (although it does lay behind some). We will use this diagram to step through the testing process using these two example intersection points.

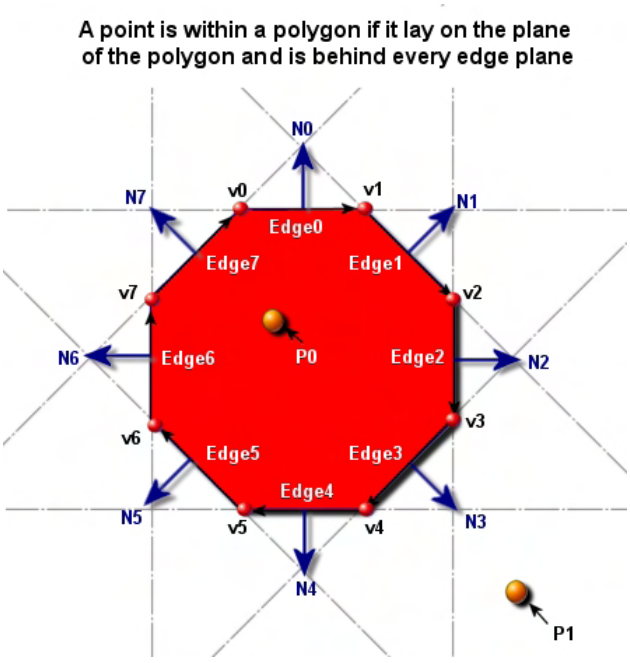


Figure 12.20

In the first example we will start with intersection point P0 which has been found to be on the plane of the polygon by a prior ray/plane intersection test. The polygon being tested for intersection is an octagon with 8 vertices labelled v0 through v7. We start by entering a loop that will iterate through each of the eight edges of the polygon. In the first iteration of this loop, we test the first edge formed by the first two vertices v0 and v1. We can create an edge vector by subtracting v0 from v1 which give us the black arrow running along the first edge in the diagram from left to right connecting those two vertices:

```
EdgeVector = v1 – v0
```

Now, while the polygon normal is not illustrated in the diagram because it is directly facing us (coming out of the page), we can imagine it is sticking out from the polygon in our direction such that it is perpendicular to the polygon. So we have an edge vector and a polygon normal at our disposal with which to generate the plane normal for the first edge (N0 in the diagram).

We know that performing the cross product between two vectors will return a third vector which is orthogonal to the two input vectors. The edge vector and the polygon normal are already perpendicular to each other, so crossing these two vectors will return vector N0 forming an orthogonal set of vectors (like a coordinate system axis). Vector N0 will be the normal to the plane for that edge. Looking at Figure 12.20 and at edge 0 in particular, we can see that the normal N0 generated by the cross product is the normal to an infinite plane aligned with the edge and shown as the gray dashed line running horizontally through that edge. We can imagine this plane to continue infinitely both to the left and right of the edge and can further imagine the plane continuing infinitely both into and out of the page on which you are viewing the diagram. Remember, these edge planes are perpendicular to the plane of the actual polygon.

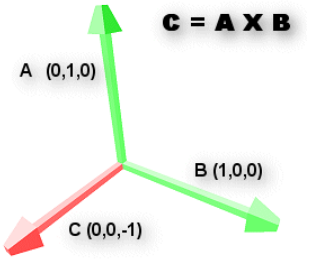


Figure 12.21

Once we have the edge plane normal for the edge we are testing, we simply classify the point P0 against that plane. If the point is in front of the edge plane, then it is clearly outside the polygon. In this example however, we can see that intersection point P0 is behind the first edge plane and we must continue through to the next iteration of the loop and check the next edge in the polygon. So, as P0 has now passed the test for edge 0, we move on to the second iteration of the edge loop where we test edge 1. Once again, we create an edge vector from the vertices of that edge (v1 and v2) like so:

```
EdgeVector =  v2 – v1
```

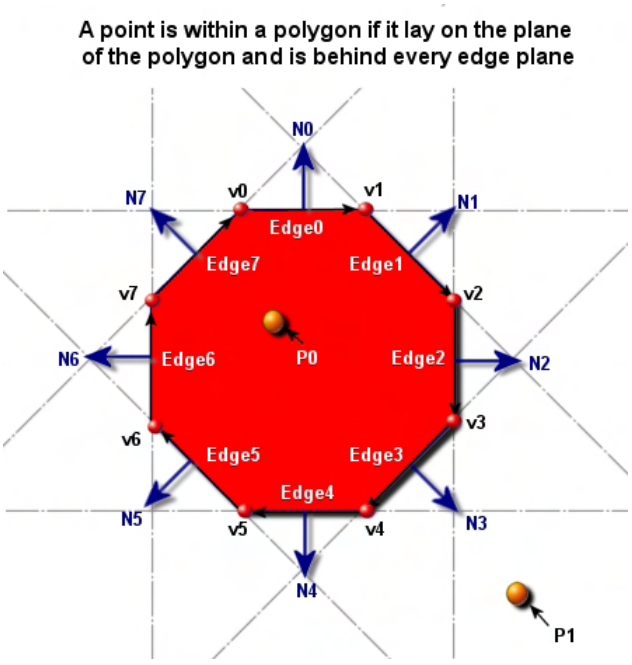


Figure 12.20 (again for convenience)

We then perform the cross product with the polygon normal and edge 1 to generate the normal N1 (the normal of the second edge). We classify the intersection point P0 against this plane and once again find it to lay behind the plane for this edge. If it lay in front of this plane we could return false from the function and skip the edge tests which have not yet been performed. In this case however, the point has passed the test for edge 1 also so we must continue to the third iteration of the loop and test the third edge.

You should be able to see by looking at the diagram that for any point that is actually inside the polygon, each edge will need to be tested and the point P0 would be found to lay behind all of the polygon's edge planes. If we reach the end of the process and have not yet returned false, it means the point is situated behind all edge planes and is therefore inside the polygon. Thus, the ray for which the intersection point with the plane was generated does indeed impact the polygon also.

Let us now see how intersection point P1 fares using this technique. We can clearly see by looking at it that it is not within the bounds of the polygon, but let us quickly walk through the testing process to see at which point it gets rejected and the process returns false.

First we create the plane for edge 0 (N0) and see that intersection point P1 is behind the plane and cannot be rejected at this point. As far as we are concerned right now, P1 may very well be inside the polygon, so we continue on to edge 1. Once again, we can see that when we test P1 against the second edge plane (edge 1), P1 is also found to be behind edge plane N1 and therefore, at this point may still be inside the polygon. When testing edge2 however, we can see that when we create the plane for this edge and classify P1 against it, P1 is found to be in front of the plane described by normal N2. As such, we can immediately return false from this procedure eliminating the need to test any more edges.

So this process does indeed correctly tell us if the point is inside the polygon, provided that we know beforehand that the point being tested is on the plane of the polygon. This all works out quite nicely because the point in polygon test is much more expensive than the ray/plane intersection test. We only have to perform the point in polygon test for a given polygon when the ray/plane intersection test was successful. When performing ray/polygon intersection tests on a large number of polygons, most of those polygons will be rejected by the cheaper ray/plane test first. This avoids the need to carry out the point in polygon test for polygons which cannot possibly intersect the ray. Once we have reduced our polygon list to a list of potential intersecting polygons, we can move to the second step and perform point in polygon tests on this subset. Note that usually only a few, at most, of the potential intersecting polygons will actually be intersected by the ray. So it is only for those polygons that every edge will need to be tested. But even the majority of these polygons tested will exit from the process early as soon as the first edge plane is found that contains the intersection point in its front space.

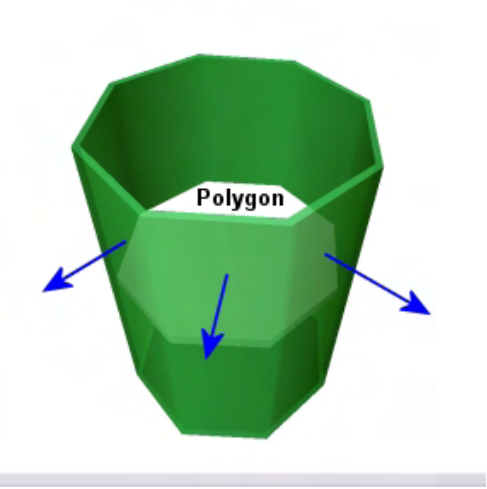


Figure 12.22

It is sometimes difficult to picture the orientations of the edge planes given the 2D image we saw earlier. But we can imagine the planes in the above example forming an infinitely long 3D cylinder based on our octagon polygon. Figure 12.22, while showing the planes as having finite length (for the sake of illustration), hopefully demonstrates this concept. The green cylinder surrounding the polygon in this example shows the planes of each edge. These planes would be infinitely tall in reality. The polygon is at the center of these planes. As we see in the diagram, the plane normals face outwards. Any point found to be behind every plane is by definition inside the space of the cylinder formed by these planes. Since we already know that the point we are testing shares the same plane as the polygon (determined by the ray/plane intersection test) we know that the point is actually on the polygon itself.

Let us now look at how we might implement a “Point in Polygon” function. We will return true if the positional vector passed in is found to be contained inside the polygon and false otherwise. Remember, this function should be used in conjunction with a ray/plane intersection test. Whenever it is called, we are always passing in a vector that describes a position on the plane of the polygon being tested. This

will be the intersection point returned from the ray/plane intersection test in a prior step. If the ray/plane test returned false for a given polygon, then there is no need to call this next function.

This simple version of the function assumes that the polygon being tested is an N-gon where all points are assumed to lay on the same plane in a clockwise winding order.

```
BOOL PointInPoly ( D3DXVECTOR3 * Point , CPolygon * pPoly)
{
   D3DXVECTOR3 EdgeNormal, Direction, Edge;
   D3DXVECTOR3 FirstVertex, SecondVertex;

   // Loop through each vertex ( edge ) in the polygon
   for ( int a = 0; a < VertexCount; a++ )
   {
       // Get First vertex in edge
       FirstVertex      = pPoly->m_vecVertices[a];

       // Get second vertex in edge  ( with wrap around)
       SecondVertex = pPoly->m_vecVertices[ (a+1)%VertexCount ] ;

        // Create edge vector
        Edge            = SecondVertex - FirstVertex;

        // Generate plane normal
        D3DXVec3Cross   ( &EdgeNormal, &pPoly->m_vecNormal, &EdgeNormal );

        //Create vector from point to  vertex ( point on plane )
        Direction       = FirstVertex-*Point;

        float d = D3DXVec3Dot( &Direction, &EdgeNormal );

        if ( d < 0 ) return FALSE;

    } // Next Edge

    return TRUE;
}
```

The code sets up a loop to test each edge (each pair of vertices). Notice how we use the modulus operator to make sure that the second vertex of the final edge wraps back around again to index the first vertex in the polygon's vertex list. We then create Edge, the edge vector of the current edge being tested.

After generating the edge vector, we create the normal for the edge plane by performing a cross product between the edge vector we just calculated and the polygon normal. In this example, we are assuming that the polygon data is represented as a CPolygon structure which contains the normal. If the normal is not present then you will have to generate it by crossing two edge vectors.

At this point we have the normal for the edge currently being tested and we also know a point that exists on that plane (i.e., either vertex in the edge). We use the first vertex in the edge, but it could be the second also, as long as the point is on the edge plane. We then calculate a vector (Direction) from the point being tested to the point on the plane (the first edge vertex) and take the dot product of this vector with the edge normal. If the angle between these two vectors is greater than 90 degrees (i.e., the result is

larger than zero) then the point is in front of the plane. As soon as this happens for any edge we know the point is not contained inside the polygon and we can exit.

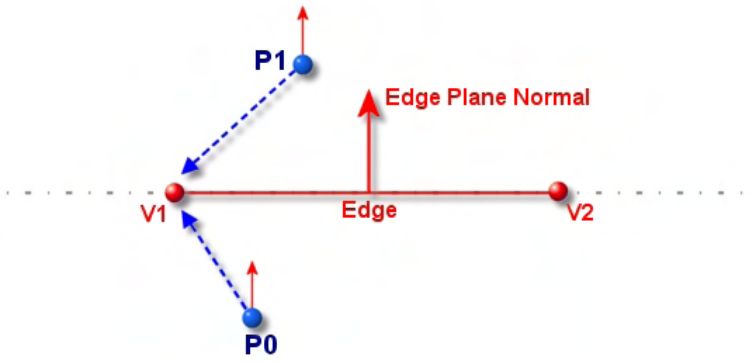


Figure 12.23

Figure 12.23 demonstrates testing to see if a point is in front or behind the edge plane. In this diagram we show two example points (P0 and P1) and perform the test on a single edge (v2 – v1). Starting with P0 we can see that creating a vector from P0 to V1 (called Direction in the above code) forms an angle with the plane normal that is less than 90 degrees. Remember that we bring vector origins together when we perform the dot product. Therefore, the dot product will return a positive result indicating that the point is behind the edge and cannot be removed from consideration without checking the other edges. The point should be tested against other edge planes as it may be inside the polygon. However, when we perform the same edge test on point P1, we can see that the vector created from P1 to V1 forms an angle with the edge normal that is larger than 90 degrees, returning a negative dot product result. When this happens, we know that the point is in front of the edge plane and cannot possibly be contained inside the bounds of the polygon. We can return false immediately and refrain from performing any more edge tests.

Bringing together everything we have learned, a function could be written to test for the intersection between a ray and a polygon by combining the ray/plane intersection test and the point in polygon test. In the following code we implement a function called RayIntersectPolygon which does just this. This function is assumed to understand the data type CPolygon which is expected to contain a polygon normal. You should be able to substitute this for your own custom polygon structures. This version of the function will return the intersection point and the intersection *t* value if the ray hits the polygon.

```
bool RayIntersectPolygon( const D3DXVECTOR3& Origin,
                          const D3DXVECTOR3& Direction,
                          const CPolygon  &Polygon,
                          float& t,
                          D3DXVECTOR3& IntersectionPoint )
{
    D3DXVECTOR3 EdgeNormal, Direction, Edge;
    D3DXVECTOR3 FirstVertex, SecondVertex;

    // Phase One : Test if Ray intersects polygon's plane
    // Calculate Distance to polygons plane using first vertex as point on plane
    float PlaneDistance = D3DXVec3Dot( &(Origin – Polygon.VertexList[0]),
                                       &Polygon.Normal );

    // Calculate Projected Ray Length onto polygons plane normal
    float ProjRayLength = D3DXVec3Dot( &Direction, &-Polygon.Normal );

    // if smaller then zero then it is either running parallel to the plane
    // or intersects the back of the plane we choose to ignore in this version.
    if ( ProjRayLength < 1e-5f ) return false;
```

```
    // Calculate t
    t = PlaneDistance / ProjRayLength ;

    // Plane is either before the start of the ray or past the end of the ray so
    // not intersection
    if ( t < 0.0f || t > 1.0f ) return false;

    // Calculate the intersection point
    IntersectionPoint = Origin + ( Direction * t );

    // Phase Two : Point in Poly
    // Loop through each vertex ( edge ) in the polygon
    for ( int a = 0; a < VertexCount; a++ )
    {
        // Get First vertex in edge
        FirstVertex     = pPoly->m_vecVertices[a];

        // Get second vertex in edge  ( with wrap around)
        SecondVertex = pPoly->m_vecVertices[ (a+1)%VertexCount ] ;

        // Create edge vector
        Edge            = SecondVertex - FirstVertex;

        // Generate plane normal
        D3DXVec3Cross   ( &EdgeNormal, &EdgeVector , &pPoly->m_vecNormal);

        //Create vector from point to  vertex ( point on plane )
        Direction       = FirstVertex-IntersectionPoint;

        float d = D3DXVec3Dot( &Direction, &EdgeNormal );

        if ( d < 0 ) return FALSE;

    } // Next Edge

    return TRUE; // Yes it must intersect the polygon

}
```

Most often in 3D graphics we are dealing with triangles and not with N-gons. The point in polygon test can be optimized somewhat when being used specifically for “Point in Triangle” tests because we can unwind the loop and lose a bit of overhead. The following snippet of code shows how we might write a function called PointInTriangle whose code could also be substituted into the function shown above to create a RayIntersectTriangle method.

A triangle will always have three vertices, so we will pass these in along with the normal for the triangle.

```
bool PointInTriangle(const D3DXVECTOR3& Point,  const D3DXVECTOR3& v1,
                     const D3DXVECTOR3& v2,     const D3DXVECTOR3& v3,
                     const D3DXVECTOR3& TriNormal )
{
    D3DXVECTOR3 Edge, EdgeNormal, Direction;
```

```
    // First edge
    Edge      = v2 – v1;
    Direction = v1 - Point;
    D3DXVec3Cross( &EdgeNormal, &EdgeNormal , TriNormal );

    // In front of edge?
    if ( D3DXVec3Dot( &Direction, &Edge ) < 0.0f ) return false;

    // Second edge
    Edge      = v3 – v2;
    Direction = v2 – Point;
    D3DXVec3Cross( &EdgeNormal, &Edge, &TriNormal );

    // In front of edge?
    if ( D3DXVec3Dot( &Direction, &EdgeNormal ) < 0.0f ) return false;

    // Third edge
    Edge      = v1 – v3;
    Direction = v3 - Point;
    D3DXVec3Cross( &EdgeNormal, &Edge, &TriNormal);

    // In front of edge?
    if ( D3DXVec3Dot( &Direction, &EdgeNormal ) < 0.0f ) return false;

    // We are behind all planes
    return true;
}
```

The cross product is not commutative so you have to make sure to feed the Edge and Plane normal vectors into the cross product in the order shown above if your edge normals point outwards. If the normals are generated so that they point inwards, then the sign of the dot product should also be changed to match. With inward facing normals, a point is considered inside the triangle if the angle formed by the vector from that point to the vertex in the edge and the edge normal is greater than 90 degrees (greater than 0.0). Alternatively, you could flip the direction of the vector from the point to the vertex and use (Point – Vertex) instead of (Vertex – Point). In that case, both vectors will have been flipped for the comparison and point into the same halfspace, so we can go back to the < 0.0 case for failure.

You will completely invalidate your results if you calculate your vectors to point in opposite directions and forget to change the sign of the dot product test. This is worth remembering since you will see instances of this in our source code from time to time.

12.4.4 Swept Sphere / Plane Intersection Testing

As discussed earlier, an application that uses our collision system will call the CollideEllipsoid method to request a position update for a moving entity. When examining a simplified version of this function, we saw the invocation of the detection phase via EllipsoidIntersectScene. We will get to study the code to this function a bit later in the chapter, but first we will concentrate on understanding the separate intersection tests which must be performed by this function in order to determine a new position for the ellipsoid.

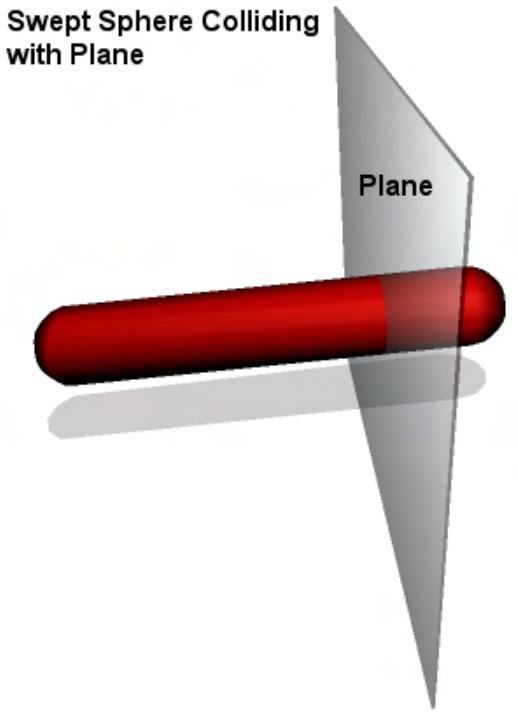


Figure 12.24

EllipsoidIntersectScene is passed the velocity vector and the ellipsoid center position in eSpace so that it can determine scene geometry intersections using unit sphere tests. The function will have to test the swept sphere against each potential colliding triangle to see if an intersection occurs. It will then return a final eSpace position based on the closest intersecting triangle (if one exists).

In order to use our unit swept sphere for intersections, this function must transform the vertices of each triangle it tests into eSpace prior to performing the intersection test. This is absolutely essential since we must have our ellipsoid and our triangles in the same space in order for the intersection results to have meaning. Over the next several sections of the lesson we will discuss the core intersection techniques that will be used by this function to determine intersections between the swept sphere and a triangle. These intersection tests will have to be executed for every triangle that is considered a potential candidate for intersection. As we currently have no broad phase in our system, this will be every triangle registered with the collision geometry database. Once we have covered the various components of the detection phase we will finally see how they are used in combination by looking at source code to the EllipsoidIntersectScene function.

When covering ray/triangle (ray/polygon) intersection, the first test we performed was ray/plane. If the ray did not intersect the plane, then it could not intersect the triangle and we can skipped any further tests. This was beneficial because this is an inexpensive test that allows us to reject most triangles very early on in the process. For the same reasons, when determining whether a swept sphere intersects a triangle, our first test is also going to be determining if the swept sphere intersects the plane of the triangle. If this is not the case then it cannot possibly intersect the actual triangle and we can bail out early from the test. Intersecting a swept sphere with a plane is going to be the cheapest intersection test we will have to perform for each triangle, so the benefit is the same for spheres as it was for rays (i.e., many of the potential colliders will be removed from consideration with very little computational overhead). Figure 12.24 demonstrates the intersection of a swept sphere and a plane. As can be seen, the sphere does indeed intersect the plane at some point along its velocity vector and we must write a function that will return that time *t* of intersection.

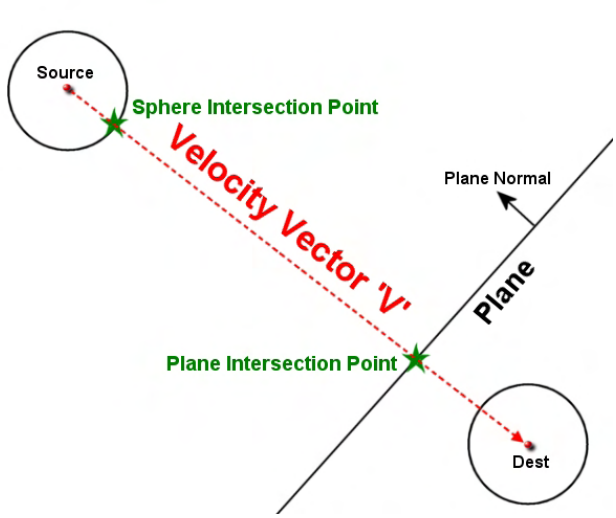


Figure 12.25

In Figure 12.25 you see the intersection we ultimately wish to find marked off. Source shows the center of the sphere in its starting position and Dest shows the sphere in its desired final position. The red dashed line is the velocity vector V describing the path we intend to move the sphere along during this update.

We want to find the *t* value for the time when the surface of the sphere first intersects the plane. The diagram shows both the intersection point on the plane and the matching point on the sphere surface that will intersect the plane first.

Unfortunately, we are not interested in finding the *t* value for the moment when the center of the sphere (Source) intersects the plane. If we were, this could be achieved by performing a simple ray/plane intersection test. You should be able to see in this diagram that if the center of the sphere Source was assumed to be the origin of a ray and the velocity vector V was the ray's delta vector, performing a ray/plane intersection test would give us the *t* value for the time when the center of the sphere would intersect the plane along the velocity vector. But this information is not helpful to us as is, because when this *t* value is later used to create the new center position of the sphere, the sphere would already be halfway embedded in the plane (and thus the triangle, assuming one exists at that location on the plane).

But as it happens, analyzing this problem sheds some light on a relationship that exists between the plane normal, the sphere radius and the intersection point on the sphere surface. By exploiting this relationship we will be able to rearrange the problem into one that allows us to use a simple ray/plane intersection test again.

In Figure 12.26 we see another example of a sphere of radius R intersecting the plane. We also see the sphere in its desired final resting place on the plane (Modified Dest). It is this center position we are attempting to find.

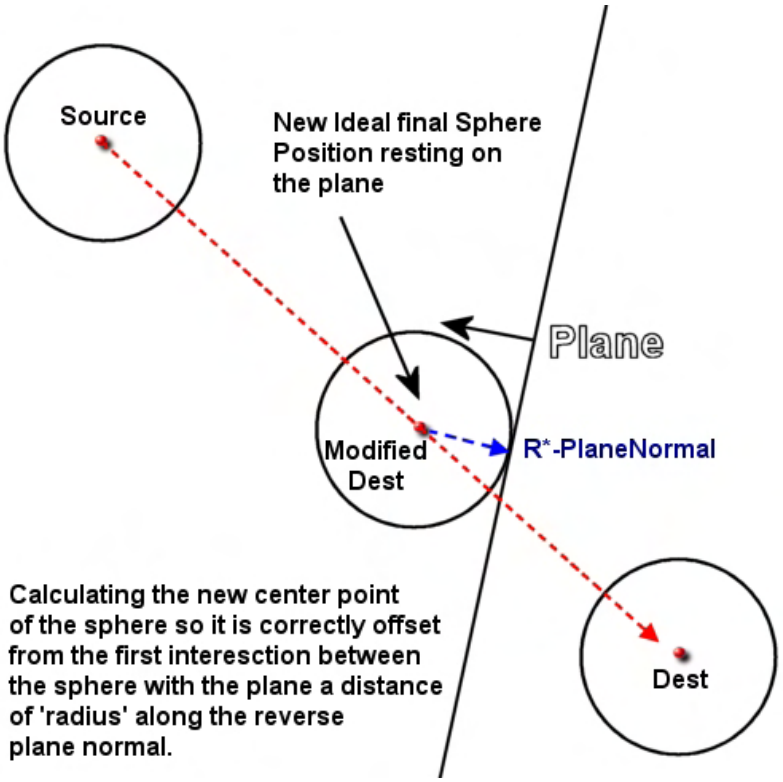


Figure 12.26

Notice that if we study the center point of the sphere resting against the plane (Modified Dest), it is offset from the plane by a distance equal to its radius. That is, it is offset along the plane normal by a distance R. Notice also how the intersection point on the surface of the sphere is found by traveling out from the center of the sphere Modified Dest by a distance of R along the negated plane normal. This is very important because it just so happens that regardless of the orientation of the plane and the orientation of the velocity vector V, if an intersection happens, the first point of intersection on the surface of the sphere will *always* be the point that is offset from the center of the sphere by a distance of R along the negated plane normal.

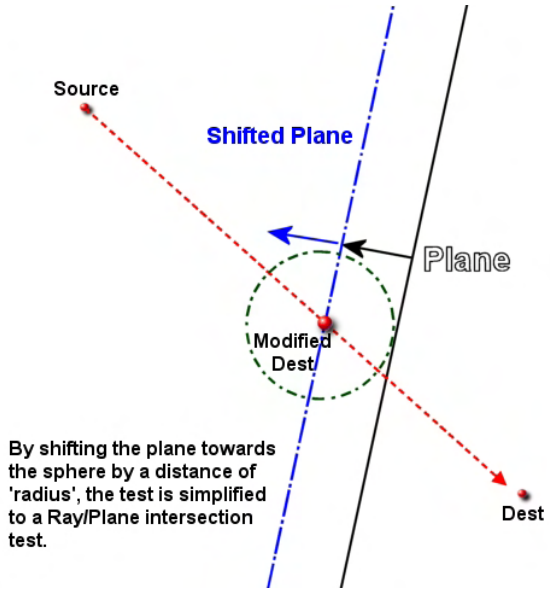


Figure 12.27

While this might not sound like a great discovery by itself, study Figure 12.26 again and imagine that you could pick up the plane and move it along the plane normal by a distance of R units. You would have essentially shifted the plane along the plane normal by a distance of –R such that the shifted plane now passes straight through the center of the sphere in its new position (Modified Dest). This is clearly shown in Figure 12.27.

As mentioned, it is the new center position of the sphere resting up against the plane we are searching for. If we shift the plane along its normal by a distance of -R, we get a plane that the center point of the sphere will intersect at exactly the same time that the surface of the sphere intersects the actual (unshifted) plane. This means we can find the new center position of the sphere by simply performing a ray/plane intersection test using the shifted plane. The ray will have the initial sphere center point

Source as its origin and it will have velocity vector V as its delta vector. As can be seen in Figure 12.27, the point at which this ray intersects the shifted plane describes the exact position where the center of the sphere should be when the surface of the sphere first intersects the actual plane. By simply shifting the plane, we have reduced the swept sphere/plane intersection test to a ray/plane intersection test (something we already know how to do very easily).

Let us put this in more formal terms:

We have the actual triangle plane defined using the plane equation (shown here in vector form):

$$P \bullet N + d = 0$$

N is the plane normal, *d* is the plane distance from the origin of the coordinate system and **P** is any point on the plane. However, we know that we are not trying to find a point on the triangle plane per se, but rather, we are trying to find the position (the center of our sphere) that is at a distance R from the plane. This is the position we would like to find since it describes the position of the center of the sphere when the surface of the sphere is touching the plane. So let us modify our plane equation to reflect this desire:

$$P \bullet N + d = R$$

For this equation to be true, the point **P** must be a distance of R from the original plane. If we subtract R from both sides of the equation, we zero out the RHS and introduce –R on the LHS:

$$P \bullet N + d - R = 0$$

As you can see, we now have the equation for the shifted plane (the original plane shifted by –R).

Next, let us substitute our ray into this equation (in place of **P**) and solve for *t*. Notice that we have placed (d – R) in brackets. This helps to more clearly identify that this is now a term describing the distance of the **shifted** plane from the origin of the coordinate system:

$$(O + tV) \bullet N + (d - R) = 0$$

where:

O = Initial Sphere Center

V = Sphere velocity vector

Now we will manipulate the equation using a bit of algebra and solve for *t*. Remember, we want to remove everything except *t* from the LHS of the equation.

$$(O + tV) \bullet N + (d - R) = 0$$

Let us first multiply out the brackets surrounding the ray by dotting the terms ***O*** and ***tV*** by ***N***:

$$O \bullet N + tV \bullet N + (d - R) = 0$$

Next we will subtract the term (*d* - *R*) from both sides, which removes it from the LHS and negates it on the RHS:

$$O \bullet N + tV \bullet N = -(d - r)$$

Now we will subtract $O \bullet N$ from both sides to remove it from the LHS. This obviously introduces the term $-O \bullet N$ on the RHS.

$$tV \bullet N = -(O \bullet N) - (d - R)$$

t is still multiplied by $V \bullet N$ on the LHS, so let us divide both sides by $V \bullet N$ leaving *t* alone on the LHS:.

$$t = \frac{-(O \bullet N + d) - R))}{V \bullet N}$$

Notice that we tidied up the numerator a little bit by moving ***+d*** from one set of negated brackets on the right into the other on the left. This way the coefficients for the original plane (the un-shifted plane) are grouped together as we are used to seeing them.

This can alternatively be written as:

$$t = -\frac{(O \bullet N + d) - R}{V \bullet N}$$

So what does this equation tell us? It tells us that all we have to do is calculate the distance from the ray origin to the triangle's plane and then subtract the sphere radius from this distance. This gives us the distance to the shifted plane in the numerator. We then divide this shifted plane distance by the length of the ray projected onto the plane normal. Of course, if you compare our final equation to the equation for the ray/plane intersection discussed in the previous section, you will see that the two are almost identical with the exception of the subtraction of R in the numerator. This small change allows the numerator to calculate the distance from the ray origin to the shifted plane instead of to the actual triangle plane.

Our swept sphere/plane intersection routine will be responsible for returning a *t* value. The calling function can then calculate the new position of the center of the sphere by performing:

New Sphere Center = Sphere Center + (t * Velocity)

where *t* is the value returned from the intersection routine we are about to write. It can be seen that because this intersection point lies on the shifted plane, it is at a distance of exactly Radius units from the actual plane and thus the sphere's surface is resting on the plane of the triangle at this time. Obviously, if the ray does not intersect the shifted plane, then the function should return false and the above step skipped.

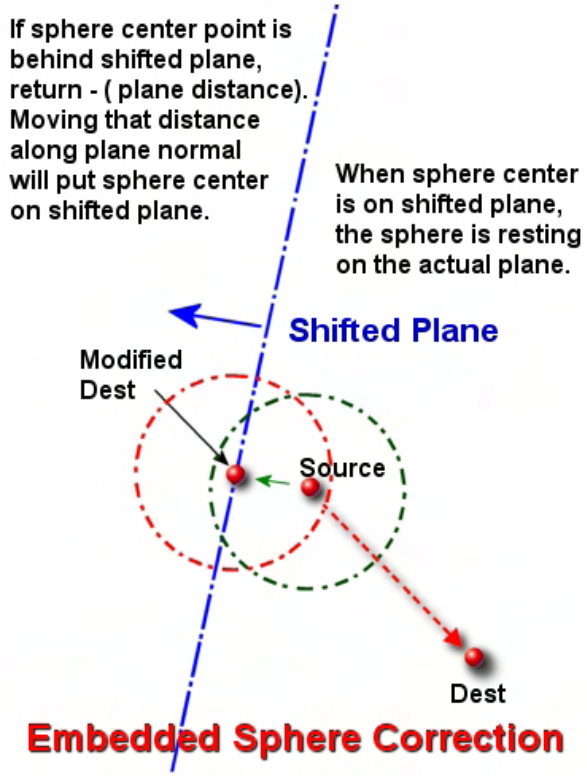


Figure 12.28

We are going to have to be vigilant about attempting to detect cases where things might not go as smoothly as planned. With floating point accumulation and rounding errors being what they are, it is possible that the sphere may have been allowed to become slightly embedded in the plane such that the ray origin actually lies behind the shifted plane of the triangle prior to the test being executed. While our collision detection system exists to stop the sphere ever passing through a triangle, there may be times when it has been moved a miniscule amount and it is partially embedded in the plane, but by a value larger than epsilon such that it is not considered to be on the plane by the intersection routines. We certainly do not want our collision detection system to fail when such an event occurs. It should instead calculate the value we need to move the sphere's center position back along the plane normal so that on function return, it is resting on the shifted plane. We will have then corrected the problem and properly moved the sphere center position back from behind the shifted plane to resting on it, as desired.

Figure 12.28 shows this additional test in action. The original center of the sphere (Source) is already behind the shifted plane when the ray intersection with the shifted plane is carried out. Usually, if the distance to the plane from the ray origin is negative in a ray/plane test, we return false for intersection. However, as Figure 12.28 demonstrates, in this particular case, we wish to calculate the distance from the center point to the plane and move it along the plane normal by that amount so that the center of the sphere is sitting on the shifted plane and we have the intersection point we were looking for. However, how do we know if the center of the sphere is legitimately behind the shifted plane of the triangle and should be corrected?

At this point we do not. Only if the intersection point with the shifted plane is later found to be within the edges of a polygon (in a separate point in polygon test) do we know if this is the case. Then we can update the position of the sphere so that it is no longer embedded. Remember, the ray origin will legitimately be behind thousands of planes as each triangle comprising the scene will lie on a plane. As planes are infinite, they will span the entire game level. We certainly do not want to move the ray origin back onto the plane for every one it is behind, as most will be nowhere near the sphere. What we will do

is narrow this down so that we only shift the ray origin back onto the shifted plane if the shifted plane itself is behind the sphere center position by a distance of no more than the radius of the sphere. In the worst cases it will never embed itself in a polygon by more than a fraction of that amount in a single update. This way we will not be performing this step needlessly for every shifted plane of every triangle which the sphere center position lays behind. Of course, we will still be needlessly moving it back for many planes since we really do not know, until we perform the point in poly tests later, whether this intersection point is within the edges of a polygon on that plane and whether the position of the sphere should really be updated in this way.

Let us now put to the test everything we have learned and examine the code to a function that performs an intersection test between a moving sphere and a plane. The following code is the exact code used in our collision system for this type of intersection. As discussed a moment ago, our intersection routines do not return the actual intersection point (a vector) but instead return only the *t* value. You will see in a moment that it is the parent function of our collision detection phase that uses the *t* value returned from its intersection routines to generate the final position of the sphere at the end of the process.

Notice first how we calculate the numerator of our equation (the distance to the shifted plane). We calculate the distance to the plane by dotting a vector from the sphere center to the point on the plane (passed as a parameter) and then subtract the radius of the sphere (just as our equation told us to do).

```
bool CCollision::SphereIntersectPlane( const D3DXVECTOR3& Center,
                                       float Radius,
                                       const D3DXVECTOR3& Velocity,
                                       const D3DXVECTOR3& PlaneNormal,
                                       const D3DXVECTOR3& PlanePoint,
                                       float& tMax )
{
    float numer, denom, t;

    // Setup equation
    numer = D3DXVec3Dot( &(Center - PlanePoint), &PlaneNormal ) - Radius;
    denom = D3DXVec3Dot( &Velocity, &PlaneNormal );

    // Are we already overlapping?
    if ( numer < 0.0f || denom > -0.0000001f )
    {
        // The sphere is moving away from the plane
        if ( denom > -1e-5f ) return false;

        // Sphere is too far away from the plane
        if ( numer < -Radius ) return false;

        // Calculate the penetration depth
        tMax = numer;

        // Intersecting!
        return true;

    } // End if overlapping

    // We are not overlapping, perform ray-plane intersection
    t = -(numer / denom);
```

```
    // Ensure we are within range
    if ( t < 0.0f || t > tMax ) return false;

    // Store interval
    tMax = t;

    // Intersecting!
    return true;
}
```

Notice the conditional code block in the middle of the function that handles the embedded case. As you can see, if the numerator is smaller than zero it means the distance to the shifted plane from the sphere center is a negative value. Also, if the denominator is larger than zero (with tolerance) it means the velocity vector is moving away from the plane and we just return false. We also return false if the distance from the ray origin to the shifted plane is larger than the radius (< -Radius). As discussed, planes that are this far away can safely be removed from consideration since the sphere could not have possibly embedded itself into a polygon this much in a single frame update. We then store the negative distance to the plane (the numerator) in the referenced parameter tMax so the caller will have access to this distance. This will describe to the caller the distance the sphere center should be moved backwards to rest on the shifted plane. In other words, it tells us how far back to move the sphere position so that it is no longer embedded in this plane.

Outside the embedded case code block we perform the usual calculation of the *t* value to generate a parametric value along the ray for the point of intersection. This is then returned to the caller via the tMax reference. Once again, in this section you can see that if t < 0.0 then we return false, since the intersection is behind the ray. Notice however that we also return false if tMax is smaller than the *t* value we just calculated. As you can see, tMax is not only an output, but it is also an input to the function.

To understand this, remember that the detection process will be run for each triangle and that we are only interested in discovering the closest intersection. Therefore, every time this function is called, it will only return true if the *t* value is smaller (closer to the sphere center point) than a previously calculated *t* value generated in a test with another triangle's plane. Thus, when we have tested every triangle, tMax will contain the *t* value for the closest colliding triangle.

Finally, if the sphere is not embedded in the plane, tMax will contain the parametric *t* value (0.0 to 1.0 range) on function return. But if the sphere *is* embedded, tMax will contain the actual distance to the shifted plane along the plane normal. The calling function will need to make sure it uses this information correctly when generating the intersection point on the plane, as shown next:

```
    if ( !SphereIntersectPlane( Center, Radius, Velocity,
                                TriNormal, v1, t )) return false;

    if ( t < 0 )
        CollisionCenter = Center + (TriNormal * -t);
    else
        CollisionCenter = Center + (Velocity * t);
```

In the above code, v1 is the first vertex in the triangle currently having its plane tested (i.e., the point on plane).

If input/output parameter *t* is negative on function return then it contains the actual distance to the plane. The point of collision on the shifted plane is calculated by moving the sphere center *t* units along the plane normal (we negate *t* to make it a positive distance). This moves it backwards so that the sphere center is now on the shifted plane and its surface is no longer embedded. If *t* is not negative then it is within the 0.0 to 1.0 range and describes the position of intersection (the new center point of the sphere) parametrically. Therefore, we simply multiply the velocity vector with the *t* value and add the result to the initial position of the center of the sphere. The end result is the same -- a position on the shifted plane which correctly describes what the position of the sphere should be such that its surface is resting on, but not intersecting, the triangle plane.

12.5 The SphereIntersectTriangle Function: Part I

Our collision system will invoke the collision detection phase by calling the CCollision::EllipsoidIntersectScene method. We will see later how this method will transform all the potential intersecting triangles into eSpace. It will then loop through each triangle and test it for intersection with the swept sphere. To test each triangle, it calls the CCollision::SphereIntersectTriangle method, which we will begin to construct in this section. This function will be a collection of different intersection techniques that we will implement. Below, we see the first part of this function using the intersection routine we discussed in the previous section.

```
bool CCollision::SphereIntersectTriangle( const D3DXVECTOR3& Center,
                                          float Radius,
                                          const D3DXVECTOR3& Velocity,
                                          const D3DXVECTOR3& v1,
                                          const D3DXVECTOR3& v2,
                                          const D3DXVECTOR3& v3,
                                          const D3DXVECTOR3& TriNormal,
                                          float& tMax,
                                          D3DXVECTOR3& CollisionNormal )
{

    float       t = tMax;
    D3DXVECTOR3 CollisionCenter;
    bool        bCollided = false;

    // Find the time of collision with the triangle's plane.
    if ( !SphereIntersectPlane( Center, Radius, Velocity, TriNormal, v1, t ))
       return false;

    // Calculate the sphere's center at the point of collision with the plane
    if ( t < 0 )
        CollisionCenter = Center + (TriNormal * -t);
    else
        CollisionCenter = Center + (Velocity * t);

    // If this point is within the bounds of the triangle,
    // we have found the collision
    if ( PointInTriangle( CollisionCenter, v1, v2, v3, TriNormal ) )
    {
        // Collision normal is just the triangle normal
```

```
        CollisionNormal = TriNormal;
        tMax            = t;

        // Intersecting!
        return true;

    } // End if point within triangle interior

    // We will need to add other tests here

}
```

This function is called for each collidable triangle and will be passed (among other things) a reference to a variable tMax which will be used to test for closer intersections. It will only return true if the *t* value of intersection (if it occurs) is smaller than the one already stored in tMax. The three vertices of the triangle (just 3D vectors) are passed in the v1, v2, and v3 parameters.

The first thing the function does is call the SphereIntersectPlane function we just wrote to test if the sphere intersects the plane of the triangle. Notice how we pass in the triangle normal and the first vertex of the triangle as the point on the plane (any vertex in the triangle will do). If the SphereIntersectPlane function returns false, then the SphereIntersectTriangle function can return false as well. If the sphere does not intersect the plane, it cannot possibly intersect the triangle on that plane. If the SphereIntersectPlane function returns true, the sphere did intersect with the plane and we continue.

The next thing we do is calculate the intersection point on the shifted plane. We can think of this position as the new position we should move the center of the sphere to so that its surface is resting on, but not intersecting, the triangle's plane. Of course, we still do know yet whether this collision point is within the triangle itself, so we must perform a Point in Triangle test. If it is inside, then we have indeed found the closest intersecting triangle so far and can return this new *t* value to the caller, which it can then use to update the sphere's position. Keep in mind that this function is called for each triangle, so we may find a triangle in a future test which is closer, overwriting the tMax value we currently have stored. We can also return the triangle normal as the intersection normal (for later use when creating the slide plane).

One thing that may have struck you as we have gone through this discussion is that we use the Point in Triangle function to test if the intersection point on the shifted plane is contained inside the polygon's edges. But the intersection point is on the shifted plane, not on the actual plane of the triangle. We can think of the intersection point as hovering above the actual polygon's plane at a distance of radius units (see Figure 12.29).

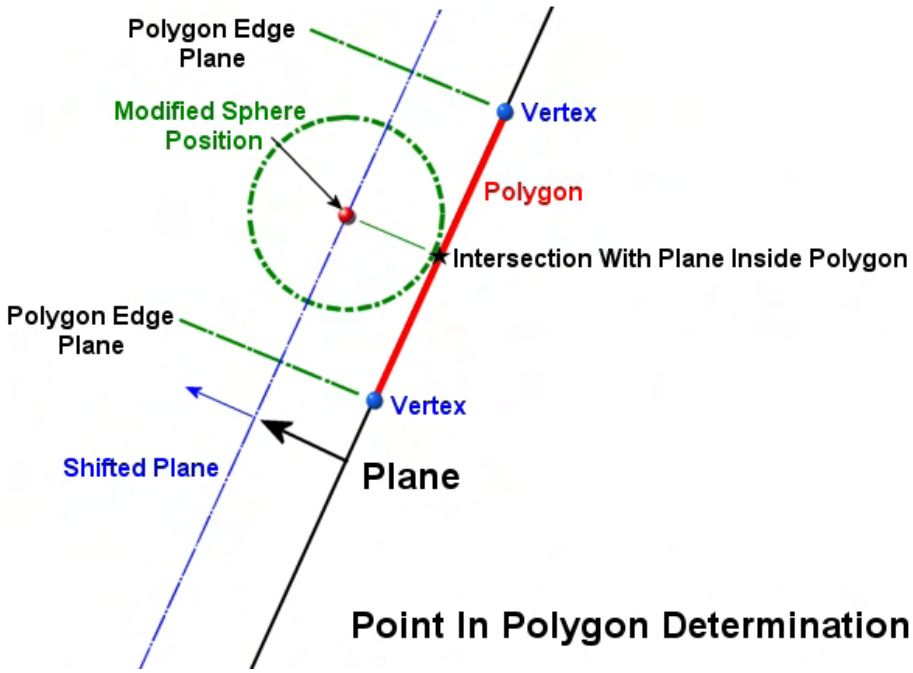


Figure 12.29

In Figure 12.29, the intersection point on the shifted plane describes the new non-intersecting position of the sphere center point. But it is not actually *on* the polygon's plane, so it raises the question as to whether or not we can use our Point in Triangle routine? The answer is yes. Recall that our Point in Polygon/Triangle routine returns true if the point is enclosed within the infinite planes formed by each edge of the polygon. Therefore, it is not necessary for the point to actually lie on the plane for the function to return true. If you refer back to Figure 12.22 you can see that a point hovering above the actual plane of the hexagonal polygon would still be inside the bounds of the infinite cylinder formed by the edge planes. In Figure 12.29, we see that the intersection point on the shifted plane (Modified Sphere Position) is indeed between the two edge planes, even though the point itself does not rest on the polygon's plane. Therefore, our Point in Triangle function would still correctly return true.

Unfortunately, our SphereIntersectTriangle method currently only handles the simplest and most common intersection case – when the sphere surface collides with the interior of the polygon. However, this is not the only case we must test for, and you will notice at the bottom of the code listing shown above we have highlighted a comment that suggests additional tests will need to be performed before we can rule out a collision between the sphere and the triangle currently being tested.

Figure 12.30 demonstrates the case where the sphere does intersect the plane and does intersect a polygon on that plane. However, the sphere does not intersect the *interior* of the polygon; it collides with one of its edges. Thus, edge tests are going to be required as well. What this essentially means is that our routine will go as follows… First we test to see if the sphere intersects the plane. If it does not, we can return false as shown in the above code. Then test to see if the intersection point on the plane is within the interior of the polygon (also shown above). If it is, then we have found the collision point and no further tests need to be performed. We simply return the *t* value of intersection. However, if the sphere did intersect the plane, but the intersection point is not within the interior of the polygon, we must test for an edge intersection (see Figure 12.30) because a collision may still occur.

Unfortunately, testing for collisions between a swept sphere and a polygon edge is not as simple as our other cases. The problem is that we can no longer rely on the fact that the first point of intersection on the surface of the sphere will be found at a distance of Radius units along the negated plane normal from the sphere center point. As Figure 12.30 demonstrates, at the time the sphere intersects the edge of the polygon, the sphere has (legally) passed through the plane. While the intersection point is still clearly at a distance of Radius units from the center of the sphere, it is no longer determined by tracing a line from the sphere center (Modified Dest) in the direction of the negated plane normal. That was how our previous test worked, but it will obviously not work here. When we call the SphereIntersectPlane function it will calculate the point at which the sphere first touches the plane. This point will not be inside the polygon, so the Point in Polygon test will return false. Thus, if the initial Point in Polygon test returns false, we must perform some more complicated edge tests before we can safely conclude that no collision has occurred.

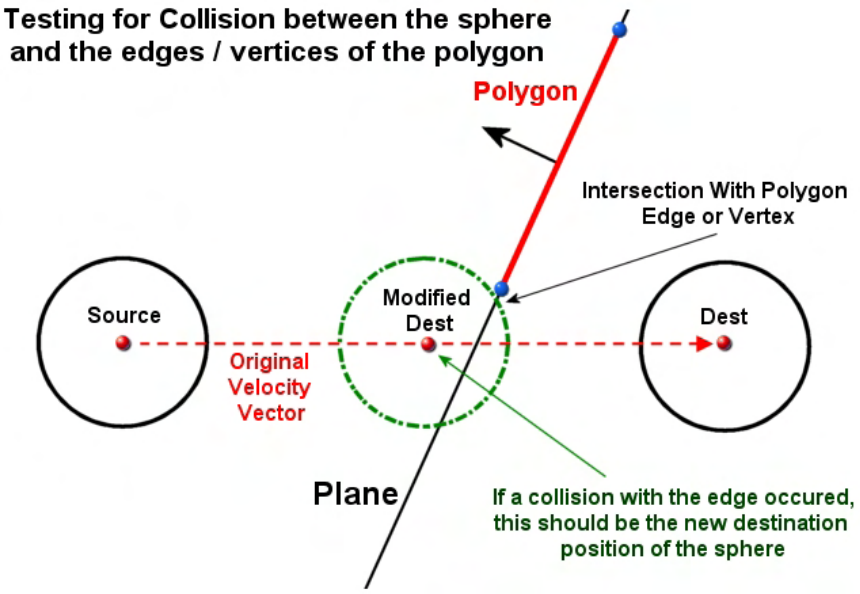


Figure 12.30

We will see later that tackling such a problem naturally leads us to a quadratic equation that will need to be solved. Because many students have not brushed up on quadratic equations since high school, the next section provides a quick refresher on quadratic equations and related ideas (including some very rudimentary mathematics like fraction multiplication and division, negative numbers, etc.). If you feel confident with respect to your ability to manipulate quadratic equations, get them into the standard form, and then solve them, please feel free to skip the next section. You should bear in mind that the remaining intersection techniques we examine will require that you have a good understanding of quadratic equations, so reading the next section is highly recommended if you are a little rusty.

12.6 Quadratic Equations

In this section we will discuss what quadratic equations are and how to solve them. If you have never encountered quadratic equations before then you will have to take a leap of faith for the time being and just believe us when we say that you will find the ability to solve them incredibly useful in your programming career. But before we get too far ahead of ourselves, we first have to discuss what a quadratic equation is and talk about a generic means for solving them. Only then will you be able to see why they are so useful in our collision detection code. Therefore, although this section will be talking about quadratic equations in an abstract manner initially, please stay with us and trust that their utility will become apparent as we progress with the lesson.

Hopefully we all recall that an equation is a mathematical expression stating that two or more quantities are the same as each other. You should also remember from high school algebra that a polynomial is a mathematical expression involving a sum of powers in one or more variables multiplied by coefficients (i.e., constants). For example, the following polynomial involves the sum of powers in one variable *t*. Polynomials are usually written such that the terms with the highest powers are arranged from left to right. This allows the degree of the polynomial to be more easily recognizable. In this example, we have a polynomial of the 3rd order because it involves a term that raises the unknown variable *t* to a maximum power of 3.

$$t^3 + at^2 + bt + c = 0$$

This is referred to as a *cubic polynomial* because the highest power of the unknown *t* is 3. **a**, **b** and **c** are referred to as *coefficients* (or constants) and contain the 'known' values, whatever they may be. Solving the above equation essentially means finding a value for *t* (i.e., the 'unknown') which makes the equation true. As you can imagine, it is no longer as easy to isolate *t* on the LHS of the equation (as it was with a linear equation such as the plane equation for example) since the unknown is now involved in multiple terms where it is being multiplied by coefficients.

A **quadratic** equation is a polynomial equation in which the highest power of the **single** unknown variable *t* is 2 and the coefficient of the squared term **a** is not zero. The **standard form** of the quadratic equation is the arrangement when the RHS is set to zero and the terms are stated as:

$$at^2 + bt + c = 0 \quad \textbf{\textit{(Standard Form)}}$$

As you can see, if **a** was set to zero, this would cancel out the squared term ($\mathbf{t^2}$) reducing the equation to a linear equation in the form:

$$bt + c = 0$$

Thus, the coefficient of the squared term must be non zero in order for it to be a quadratic equation.

> **Note:** The name 'Quadratic' given to such an equation is often the subject for much confusion. We know that the word 'Quad' means four and yet it is used to describe an equation which involves the 2nd power

and not the 4th. But there is another meaning for the word 'quad' which helps us make sense of why an equation of this type is called a Quadratic. Although the Latin word 'Quadri' does indeed mean four, the word 'Quadrus' means 'a square' -- a shape that has four sides. Quadratic equations arose from geometry problems involving squares, and as we know, the second power is also referred to as a square. Another Latin word 'Quadratus' means 'Squared', and several other words derive from this. For example, another name for the 'Square Dance' is the 'Quadrille' and the word 'Quadrature' (through the French language) describes the process of constructing a square of a certain area. Therefore, 'Quadratic' refers to an equation involving a single variable in which the highest power is raised to the power of two and is therefore said to be squared.

Not all quadratic equations are readily arranged into the shape the of the standard form $at^2 + bt + c = 0$. For example, the following list shows some quadratic equations which look initially to be using very different shapes. Notice however, that the single unknown (**x**) is still raised to the second power in one of the terms.

$$x^2 = 81$$

$$6x^2 + x = 100$$

$$-x^2 - 100 = 20$$

$$(x+3)^2 = 144$$

While these equations are all quadratic equations in **x**, they are not currently in the standard form. Later we will discuss the *quadratic formula*. This is a technique which can always be used to find the solutions to a quadratic equation that has been arranged in the standard form. Thus, in order to use the quadratic formula, we must first apply some algebraic manipulation to a quadratic equation to get it into the shape of the standard form $at^2 + bt + c = 0$. In the standard form, information we know about is stored in the coefficients **a**, **b** and **c** and *t* is the value we are trying to find. We need to be able to feed the correct **a**, **b** and **c** values into the quadratic formula so that it can be used to solve for the unknown *t* (**x** in the above list of examples).

Let us see how the above equations could be arranged into the standard form so that we could solve them using the quadratic formula. For now, do not worry about what the quadratic formula is or how we solve the above equations; for the moment, we are just going to focus on taking a quadratic equation in various forms and manipulating it into the standard form. Later on we will address the quadratic formula and find out how it can be applied to solve such equations.

Example 1: Representing $x^2 = 81$ in standard form.

The standard form has zero on the RHS of the equals sign, so let us first subtract 81 from both sides of the equation to make that true. Our equation now looks like this:

$$x^2 - 81 = 0$$

Notice that in our equation **x** is used as the label for the unknown, while in the standard form ***t*** is generally used. This really is just a mater of preference, so you can think of **x** in our equation as representing ***t*** in the standard form. Sometimes, **x** is also used to represent the unknown in the standard form. As you recall, in algebra, these are all just placeholders, so we can use whatever letters we wish (although it is helpful to follow convention, since it makes it easier for others to pick up on what is happening).

Now, if we look at the standard form $at^2 + bt + c = 0$, you should see that we essentially have what we are looking for. We know that in our current equation the squared term is not multiplied by anything (it has no **a** coefficient like the standard form). We also know that this is equivalent to having an **a** coefficient of 1 in the standard form.

$$1x^2 - 81 = 0$$

So we can see that **a=1** in this example. Our equation also has no **bt** (**bx**) term like the standard form does. This is equivalent to having a **b** coefficient of zero. Obviously, having a **b** value of zero would cancel out any contribution of the middle term, as shown below.

$$1x^2 + 0x - 81 = 0$$

We now have our equation in the standard form where **a**=1, **b**=0 and **c**= -81 as shown below. Compare it to the standard form:

$$1x^2 + 0x - 81 = 0$$

$$at^2 + bt + c = 0$$

If we wanted to use the quadratic formula to solve this equation we could simply feed in the values, **a**=1, **b**=0 and **c**=-81. Of course, $\mathbf{1x^2}$ is the same as just $\mathbf{x^2}$, and in the middle term, **0x** evaluates to nothing and the term is completely cancelled out. This can also be written as follows, which is the original equation we had:

$$x^2 - 81 = 0$$

or

$$x^2 = 81$$

To be sure, this is a very simple quadratic equation to solve, even without using the quadratic formula. But if you wish to solve it using the quadratic formula, you could use the inputs **a**=1, **b**=0 and **c**=-81, as we have just discovered.

Example 2: Representing $6x^2 + x = 100$ in standard form.

First, let us subtract the 100 from both sides of the equation to leave zero on the RHS as the standard form dictates:

$$6x^2 + x - 100 = 0$$

This example is actually a little easier to spot the relationship with the standard form. The standard form starts with at^2 and therefore we can see right away that **a** = 6. We can also see that the second term of the standard form is bt (***bx*** in our equation). Since bx is the same as x if **b** = 1, we know that the lack of a **b** coefficient in our current equation's middle term must mean that **b** = 1. We can also see that the removal of 100 from the RHS has introduced a **c** coefficient of –100 on the LHS, giving us the values **a**=6, **b**=1 and **c**= -100 in the standard form:

$$6x^2 + 1x - 100 = 0$$

$$at^2 + bt + c = 0$$

Therefore, we could solve the quadratic equation $6x^2 + x = 100$ by plugging the values **a=6**, **b=1** and **c=-100** into the quadratic formula to solve for **x**.

Example 3: Representing $-x^2 - 100 = -60$ in standard form.

Once again, the first thing we do is add 60 to both sides of the equals sign to leave us with zero on the RHS, as the standard form dictates.

$$-x^2 - 100 + 60 = 0$$

Currently, we have a **c** coefficient of (-100+60) and we can instantly simplify by adding +60 to –100 to get an equation with an equivalent **c** coefficient of -40:

$$-x^2 - 40 = 0$$

The lack of any middle term *bt* (or in our case, **bx**) means that **b** = 0, canceling that term out altogether. The lack of any coefficient next to the squared term must also mean **a** = -1 (as $-1*x^2$ is just $-x^2$). Therefore, in the standard form, our equation looks like the following (once again, shown alongside the standard form template for comparison):

$$-1x^2 + 0x - 40 = 0$$

$$at^2 + bt + c = 0$$

Thus, in order to solve our original equation $-x^2 - 100 = -60$ using the quadratic formula, we would feed in the values **a**=-1, **b**=0, **c**=-40.

Example 4: Representing $(x+3)^2 = 144$ in standard form.

This example looks quite different from any we have seen so far. Is this still a quadratic equation? Well the answer is yes even though it is being shown in its squared form on the LHS. To understand why it is indeed a quadratic, let us see what happens to the LHS if we multiply out the square. This will bring to our attention two **very important** algebraic identities.

We know that if we have the term i^2, then the term can be evaluated by performing $i * i$. You could say that we have multiplied the square out of the term, since we now have the value of $i * i$ and no longer have a power in that term. So how do we multiply the square out of a term like $(o+e)^2$? Well, we just treat the content inside the brackets (the squared term) as the single term '*i*'. Therefore, instead of doing $i * i$, we do:

$$(o+e)(o+e)$$

The next question concerns how we multiply the expression (o+e) by the expression (o+e). Hopefully most of you will remember learning about multiplying bracketed terms in high school. In particular you may remember the acronym FOIL. FOIL is used as an aid to remember the multiplication order for bracketed terms. The acronym stands for the words "**F**irst **O**uter **I**nner **L**ast and describes the paired terms that have to be multiplied. Let us do what it says and see what happens.

First we multiply the First terms in each bracket:

(o + e) (o + e) = o * o = o^2

Next we multiply the Outer terms of each bracket:

(o + e) (o + e) = oe

Then we multiply the Inner terms of each bracket:.

(o + e) (o + e) = eo (which is the same as **oe)**

And finally, we multiply the Last terms in each bracket:

(o + e) (o + e) = e^2

Adding the results of each separate multiplication together, we get the final expanded version of $(o+e)^2$:

$$o^2 + oe + eo + e^2$$

Since **oe** and **eo** are equivalent, we are just adding this value to the expression twice. Therefore, we can collect those terms and rewrite as follows:

$$o^2 + 2eo + e^2$$

And there you have it. We have expanded the original squared term out of its brackets. We have not changed the value of the expression in any way, but we have now discovered one very important algebraic identity. That is, we have found an equivalent way of writing the expression $(o+e)^2$. The three term polynomial $o^2 + 2eo + e^2$ is referred to as being a *perfect square trinomial* (PST) as it can instantly be written in its 'easier to solve' squared form of $(o+e)^2$.

Identity 1: $(o+e)^2 = (o+e)(o+e) = o^2 + 2eo + e^2$

We will see later when deriving the quadratic formula that this identity is very important. Remember it and make sure you understand it. It will become critical for you to know that when you have a LHS of an equation that looks like this: $o^2 + 2eo + e^2$, that it can also be written like this: $(o+e)^2$ without changing its evaluation. Being able to recognize patterns like this will help you understand how quadratic equations are solved and how to manipulate quadratic equations into the standard form.

Another very important identity that is used often in solving quadratic equations is very similar to the last one except it has a negative in the initial bracketed expression:

$$(o-e)^2$$

As we know, this is equivalent to multiplying the term (o-e) by itself:

$$(o-e)(o-e)$$

FOIL can once again be applied to multiply out the brackets.

First we multiply the First terms in each bracket.

(o - e) (o - e) = o * o = o^2

Next we multiply the Outer terms of each bracket. Here we are multiplying positive **o** with negative **e**. A positive times a negative gives us a negative, so:

(o - e) (o - e) = -oe

Now we multiply the Inner terms of each bracketed expression. This time we are multiplying negative **e** with positive **o** and once again we get a negative:

(o - e) (o - e) = -eo (which is the same as **-oe)**

Finally, we multiply the Last terms in each bracket. This time we are multiplying negative **e** by negative **e** which we know produces a positive:

(o - e) (o - e) = e^2

Adding the results of each separate multiplication together we get the final expanded version of $(o-e)^2$:

$$o^2 - oe - eo + e^2$$

Since **oe** and **eo** are equivalent, we are essentially subtracting **oe** from the expression twice, so we can collect those terms and rewrite as:

$$o^2 - 2eo + e^2$$

This is another very important algebraic identity. Learn it and remember it. It is important to recognize the shape of an expression such as $o^2 - 2eo + e^2$ and know that it can also be written as $(o-e)^2$. That is, $o^2 - 2eo + e^2$ is also a perfect square trinomial (PST) that can instantly be represented in its 'easier to solve' squared representation $(o-e)^2$.

Identity 2: $(o-e)^2 = (o-e)(o-e) = o^2 - 2eo + e^2$

Now we will get back to the task at hand and determine how we can represent our final example equation $(x+3)^2 = 144$ in standard form.

First, we know from the algebraic identities we just discovered that $(x + 3)^2$ can also be rewritten as shown in Identity 1. In our example, x is equivalent to the o variable in the identity and 3 is equivalent to the e variable.

Therefore, if

$$(o+e)^2 = (o+e)(o+e) = o^2 + 2eo + e^2$$

then

$$(x+3)^2 = (x+3)(x+3) = x^2 + 2x3 + 3^2$$

Our equation now looks as follows, when compared to the standard form:

$$x^2 + 2x3 + 3^2 = 144$$

$$at^2 + bt + c = 0$$

We are not quite there yet. However, our middle term (**2*x*3**) can also be written as **2*3*x**, which is the same as **(2*3)*x**. So we can simplify our term to **6x**. We also have no **a** coefficient, which means it is equivalent to having an **a** coefficient of 1. So let us reorder things a little and see what we get:

$$1x^2 + 6x + 3^2 = 144$$

$$at^2 + bt + c = 0$$

Nearly there! We currently have a **c** term of 3^2, which evaluates to 3*3=9:

$$1x^2 + 6x + 9 = 144$$

$$at^2 + bt + c = 0$$

The only difference now between the current form of our equation and the standard form is that we have a value of 144 on the RHS and the standard form has a value of zero. Therefore, we must subtract 144 from both sides of the equation so that we are left with 0 on the RHS:

$$1x^2 + 6x + 9 - 144 = 0$$

When shown along side the standard form we can see that we currently have a **c** term of (+9-144). Therefore, our **c** coefficient evaluates to 9-144 = -135. Below, we see the final version of our equation in the standard form.

$$1x^2 + 6x - 135 = 0$$

$$at^2 + bt + c = 0$$

If we wanted to use the quadratic formula to solve the quadratic equation $(x+3)^2 = 144$, we would feed in the known values: **a=1 , b=6 c=-135.**

This may all sound very abstract at the moment, since we are simply manipulating an equation into the standard form. But the standard form of the quadratic equation is very important to us because if we can get the quadratic equation we need to solve into this shape, we can use the quadratic formula to solve it.

Thus, if we wish to solve a problem that naturally gives rise to a quadratic equation, and we can manipulate the information we *do* know about the problem into the **a**, **b** and **c** coefficients, then we can use the quadratic formula to solve for the unknown *t*. This means we can use the quadratic formula to solve *any* quadratic equation in exactly the same way, so long as we mold the problem into the shape of the standard form first.

We shall see later that our collision system must perform an intersection test between a ray and a sphere. The equation of a sphere (which belongs to a family known as Quadric Surfaces) involves squared terms and as such, when we substitute the ray equation into the sphere equation of the sphere to solve for *t* (time of intersection along the ray), this naturally unravels into an equation where the unknown *t* is in a squared term. Therefore, we will have ourselves a quadratic equation that needs to be solved. When this is the case, we can manipulate the equation into the standard form and then solve for *t* using the quadratic formula giving us the time of intersection along the ray. Essentially, solving a quadratic equation boils down to nothing more than isolating t^2 on the LHS and finding the square roots of that term. We will learn more about this later.

12.6.1 Solutions to Simple Quadratic Equations

The quadratic formula we will study later in this lesson is not the only method for solving quadratic equations. Other approaches include *factoring* and a method called *completing the square*. Since factoring will not work with all quadratic equations, we will not be using this method. Instead we will concentrate on the methods that work reliably in all circumstances. It is also worth noting that we do not have to manipulate a quadratic equation into the standard form to solve it; we only need to do this if we wish to use the quadratic formula. Many quadratic equations are simple to solve and as such, manipulating them into the standard form and then using the quadratic formula would be overkill. So before we look at how to use the quadratic formula and examine its derivation, we will look at some simple quadratic equations that can be solved in a very simple way. This will allow us to examine the properties of a quadratic equation and the solutions that are returned.

Example 1: $x^2 = 81$

We examined this exact equation earlier and discussed how to represent it in standard form so that the quadratic formula could be used to solve it. Of course, this quadratic is extremely simple to solve without manipulation into standard form and indeed using such a technique would certainly be taking the long way round.

We know that we intend to find the value of x and we currently have x^2 on the LHS. We also know that the square root of x^2 is simply x. Therefore, to isolate x on the left hand side, we can just take the square root of both sides of the equation. Remember, it is vitally important that whatever you do on one side of the equals sign must be done on the other side too.

Step 1: Original equation is in its squared form

$$x^2 = 81$$

Step 2: Remove the squared sign from x by taking its square root. Remember to take the square root of the RHS as well.

$$\sqrt{x^2} = \sqrt{81}$$

Step 3 : The square root of x^2 is x and the square root of 81 is 9, so we have:

$$x = 9$$

This is correct, but we have forgotten one important thing. Every positive real number actually has two square roots -- a positive root and a negative root. If it has been a while since you covered this material in high school then that might come as a surprise. After all, if you punch 81 into a calculator and take its square root, you will have one answer returned: 9. Likewise, if you use the C runtime library's sqrt function, it too returns a single root. In both instances, we are getting the positive root of the number. However, we must remember that when we multiply a negative number by a negative number, we get a positive number. Thus, when we square a negative number we get a positive result. This leads to the conclusion that a positive number must have a negative square root as well.

For example, we know that 9 is the square root of 81 because:

$$9^2 = 9 * 9 = 81$$

Clearly 9 is the square root of 81 because 9 squared is 81. But as mentioned, this is actually only one of two square roots that exist (the positive root). –9 is also a square root of 81 because –9 multiplied by itself is 81:

$$-9^2 = -9 * -9 = 81$$

So, we can see that the square roots of 81 are in fact, 9 and –9. This means there are actually two possible solutions for **x** in our equation: 9 and –9. This duality is often indicated using the $\pm$ sign which is basically a plus and a minus sign communicating that there is a negative and positive solution to the square root. Therefore, we should correctly show our solution as:

$$x^2 = 81$$

$$x = \pm\sqrt{81}$$

$$x = \pm 9$$

Thus, there are two answers: **x =** +9 and **x =** –9.

The fundamental theorem of algebra tells us that because a quadratic equation is a polynomial of the second order, it is guaranteed to have two roots (solutions). However, some of these may be degenerate

and some of them may be complex numbers. We are only interested in real number solutions in this lesson, so we will consider a quadratic that returns two complex roots as having no roots (and thus no solutions for the equation). If you have never heard of complex numbers, do not worry because we are not going to need to cover them to accomplish our goals. However, you can learn more about complex numbers in our Game Mathematics course if you are interested in pursuing the subject further.

Example 2: $x^2 + 25 = 25$ (A Degenerate Root Example)

This example is interesting because it present a case where only one root exists. Actually, when this happens, we can perhaps more correctly say that two roots are returned, but both the negative and positive roots are the same. Let us solve this equation step by step.

Step 1: Let us first isolate x^2 on the LHS by subtracting 25 from both sides of the equation.

$$x^2 = 0$$

Step 2 : Now to find x we simply take the square root of both sides as shown below.

$$x = \pm\sqrt{0}$$

Now you can probably see where we are going with this. The solutions for x (the roots) are the negative and positive square root of zero. Since the square of zero is always just zero, negative zero and positive zero is just zero. Therefore, we do have two roots, but they are exactly the same. If ever we get to a point when solving a quadratic equation where we have isolated x^2 on the LHS and have zero on the RHS, we know we have a situation where we have multiplicity of the root and both roots evaluate to zero.

Example 3: $x^2 + 25 = 0$ (An example with no **real** solutions.)

This example demonstrates the case where no real numbered solutions can be found for the equation. You can probably spot why this the case right away just by looking at the equation, but if not, let us try to solve it and see what happens.

Step 1: Isolate x^2 on the LHS by subtracting 25 from both sides of the equation. This will cancel out the '+25' term on the LHS.

$$x^2 = -25$$

Step 2: Now we have to take the square root of both sides of the equation to get x isolated on the LHS.

$$x = \pm\sqrt{-25}$$

And here in lies our problem. We cannot take the square root of a negative number and expect a real number result. Just try punching a negative number into a calculator and hitting the square root button; it should give you an error. This is actually pretty obvious when only a few moments ago we said that multiplying a negative by a negative gives us a positive. Therefore, if squaring a negative number gives us a positive number then there can be no real number such that when it is squared results in a negative number. Therefore, a negative number cannot possibly have real square roots. This is actually one of the main reasons that complex numbers were invented. They actually fix this problem through the use of something called imaginary numbers. However, we are not interested in complex numbers in this lesson so we will consider any quadratic equation for which real roots cannot be found, to have no solutions. This is detectable because we know this situation arises as soon as we have x^2 on the LHS and a negative number on the RHS.

Example 4: $(x+3)^2 = 144$

This is also an interesting example because it is the first we have tried to solve in which the LHS is a squared expression instead of a single value. It just so happens that a quadratic equation such as this is very simple to solve. Let us go through solving it step by step.

Step 1: The expression x+3 is currently squared on the LHS, so we wish to remove the square. We do this by simply take the square root of both sides of the equation. The square root of $(x+3)^2$ is simply x+3 (we just remove the square sign) and the square root of the RHS (144) is ±12 .

$$x+3 = \pm\sqrt{144}$$

Step 2: This evaluates to:

$$x+3 = \pm 12$$

Step 3: **x** is still not isolated on the LHS, so we can fix that by subtracting 3 from both sides of the equation:

$$x = -3 \pm 12$$

Therefore, the two solutions for **x** are

$$x = -3+12 = 9$$

and

$$x = -3-12 = -15$$

So:

x = *9* and ***x*** = *-15*

At this point you may be wondering how we know which solution is the one we actually need to use. For example, when we solve a quadratic equation to find a *t* value along the ray at which an intersection occurs with a sphere, we will only be interested in the positive solution. If we think of an infinite ray passing through the center of the sphere however, we can see how it will intersect with the surface of the sphere twice; once on the way in and once again on the way back out. In fact, this is a property that defines a surface as being a quadric surface (a family of surfaces to which spheres, ellipsoids and cylinders belong).

If we imagine stepping along a ray, we would be interested in using the smallest solution, since this will be the first point of intersection along that ray. That is to say, we are interested in the time the ray first intersects the sphere. We will generally have no interest in the second solution which gives a *t* value for the ray passing out the other side of the sphere.

However, it is worth noting that we are interested in the smallest positive root. A negative *t* value describes an intersection occurring behind the ray origin. If we imagine a situation where our ray origin is inside the sphere to begin with, we will have a positive and a negative solution. The positive solution will be the point at which our ray intersects the far side of the sphere, and the negative solution will describe the distance to the surface backwards along the ray. However, it really does depend on what you are trying to do and the quadratic you are trying to solve. For example, you may well be interested in finding both intersection points with an infinite ray and a sphere. However, if the above example was returning intersection distances between our ray and the sphere surface, our collision system will only be interested in using the positive solution and we will discard information about intersections that have occurred behind the ray origin (more on this later).

Example 5: $x^2 + 6x + 9 = 144$ (Solving a Perfect Square Trinomial)

This example looks a lot harder to solve than the previous one where the LHS was already represented as a squared expression. Now you will find out why we discussed those two algebraic identities earlier in such detail and why it is vital that you recognize them.

Take a look again at Identity 1 that we discussed earlier:

Identity 1: $o^2 + 2eo + e^2 = (o + e)^2$

Now let us look at $o^2 + 2eo + e^2$ along side the LHS of our equation.

$$o^2 + 2eo + e^2$$
$$x^2 + 6x + 9$$

We can see that **x** in our equation is equivalent to **o** in the identity. We can also see that **2e** in the identity is equal to **6** in our equation. Of course, if **2e = 6** then **e** must equal 3. The identity has $\mathbf{e^2}$ as its third term and we know that in our equation **e = 3** and that $\mathbf{3^2 = 9}$. So our equation perfectly matches the shape of $(o + e)^2$ when expanded out of its brackets. Thus, our equation is a perfect square trinomial that can be readily written in its squared form:

If

$$o^2 + 2eo + e^2 = (o + e)^2$$

then

$$x^2 + 6x + 9 = (x + 3)^2$$

This means that our original equation:

$$x^2 + 6x + 9 = 144$$

can also be written as

$$(x + 3)^2 = 144$$

While this equation initially looked quite different from the last example, we now see that it is exactly the same equation. We can prove this by performing FOIL on $(x+3)^2$:

First we multiply the First terms in each bracket.

(x + 3) (x + 3) = x * x = x^2

Next we multiply the Outer terms of each bracket

(x + 3) (x + 3) = 3x

Now we multiply the Inner terms of each bracket.

(x + 3) (x + 3) = 3x

And finally, we multiply the Last terms in each bracket.

(x + 3) (x + 3) = 9

Adding the results of each separate multiplication together we get the final expanded version of $(x + 3)^2$:

$$x^2 + 3x + 3x + 9$$

As we are adding **3x** to the LHS expression twice, we can collect these like terms and rewrite the LHS as the following:

$$x^2 + 6x + 9$$

As you can see, this is the original LHS of the equation we started with.

Now that we have identified (and re-checked using FOIL) that we can write the LHS in its squared form, solving the equation becomes no different from the previous example with the steps shown below.

Step 1: Our original equation.

$$x^2 + 6x + 9 = 144$$

Step 2: Identify that the equation is a PST (perfect square trinomial) and can be readily written in squared form on the LHS.

$$(x + 3)^2 = 144$$

Step 3: Take the square root of both sides of the equation, removing the power from the LHS.

$$x + 3 = \pm\sqrt{144}$$

Step 4: This evaluates to + and – 12 on the RHS

$$x + 3 = \pm 12$$

Step 5 : Subtract 3 from both sides to isolate **x** on the LHS

$$x = -3 \pm 12$$

This yields two solutions for **x**:

$$x = -3 + 12 = 9 \quad \text{and} \quad x = -3 - 12 = -15$$

In this example we have illustrated the importance of recognizing when the LHS can be written in its squared form. This boils down to nothing more than pattern matching if your LHS is already a PST. As long as you can make this connection with the two algebraic identities discussed earlier, you can immediately represent the LHS of the equation in its much simpler squared form. This makes it much easier to find the solution and isolate the unknown.

12.6.2 Completing the Square

We have seen that it is easy to find the solution to a quadratic equation that is in the form $(x+3)^2$ on the LHS. We have also demonstrated how the LHS of a quadratic equation in the form $o^2 + 2eo + e^2$ is a perfect square trinomial and can be written as $(o+e)^2$ and easily solved. For example, we identified that the LHS of an equation in the form of $x^2 + 6x + 9 = 144$ can readily have its LHS written as $(x+3)^2 = 144$ because the LHS represents a polynomial that can be readily written as a squared expression.

The next question is, how do we solve an equation whose LHS is not a perfect square trinomial and therefore cannot be readily written as an expression in squared form? For example, how would we a find the solutions to a quadratic equation whose LHS is in the form: $2x^2 + 12x$. This equation's LHS is clearly not a PST in the form of $o^2 + 2eo + e^2$ and thus can not be written in $(o+e)^2$ form.

Example 6: $2x^2 + 12x = 0$ (Solving a Quadratic Equation that is not a square.)

A quadratic equation in this form is not so easy to solve because the LHS is not a perfect square. Therefore, we have to manipulate the equation so that the LHS becomes a perfect square trinomial allowing us to then represent it in its squared form. This process is known as **Completing the Square**.

> **Note:** Many sources cite the Babylonians as the first to solve quadratic equations (around 400BC). However, the Babylonians had no concept of an *equation*, so this is not quite fair to say. What they did do however was develop an approach to solving problems algorithmically which, in our terminology, would give rise to quadratic equations. The method is essentially one of completing the square. Hindu mathematicians took the Babylonian methods further to give almost modern methods which admit negative quantities. They also used abbreviations for the unknown, usually the initial letter of a colour was used, and sometimes several different unknowns occurred in a single problem.

In order to be able to represent the LHS in Example 6 in its squared form, we must somehow get it into a form that makes it a perfect square. That is to say, we must try to mold it into the form: $o^2 + 2eo + e^2$.

Let us see step by step how the process of completing the square is performed. We will start by showing the LHS of our equation alongside the form of the perfect square trinomial we are trying to mold it into:

$$o^2 + 2eo + e^2 = 0$$

$$2x^2 + 12x = 0$$

Step 1: In our equation **x** is the unknown, which corresponds to **o** in the square trinomial. We also have an **x** coefficient of 2. In the square trinomial the $\mathbf{o^2}$ term is on its own (it has no coefficient) so we want

to get $\mathbf{x^2}$ on its own also. Since **x** is currently multiplied by 2 in our equation, we can cancel it out by dividing the equation (both the LHS and the RHS) by 2:

$$x^2 + \frac{12}{2}x = \frac{0}{2}$$

As you can see, dividing $\mathbf{2x^2}$ by 2 yields $\mathbf{x^2}$, removing the coefficient of the $\mathbf{x^2}$ term as we intended. Actually, we could say that the $\mathbf{x^2}$ term now has an implied coefficient of 1. We must also divide every other term of the equation by 2 so that the equation stays balanced. We divide the term **12x** by 2 reducing the term to **6x.** The term on the RHS is divided by 2 also, but as it is currently is set to zero, dividing by anything will still result in a zero and as such, the RHS remains unchanged. This evaluates to the following equation (shown again along side the trinomial whose form we are trying to achieve):

$$o^2 + 2eo + e^2 = 0$$
$$x^2 + 6x = 0$$

Step 2: Remembering that **x** in our equation is equivalent to **o** in the PST, we can see that we have correctly reduced the $\mathbf{x^2}$ term on the LHS so that it has no coefficient (or an implied coefficient of 1). This is consistent with the $\mathbf{o^2}$ term in the PST. We can also see that in the middle term of the square trinomial, the unknown **o** has a coefficient of **2e.** In the middle term of our equation, **x** has a coefficient of 6. Therefore, we can say that in our equation **2e = 6** and thus, **e = 3**. All we have to do to make our equation a PST is add $\mathbf{e^2}$ to both sides of the equation (LHS and RHS must remain balanced). We have just discovered that if **2e = 6** then **e = 3** and thus $\mathbf{e^2 = 3*3 = 9}$. So, in this step we add 9 to both the LHS and the RHS of the equation, resulting in the following:

$$x^2 + 6x + 9 = 9$$

Now we have a perfect square trinomial where **o** is equivalent to **x** and **e** is **3**. Identity 1 tells is that the LHS of this equation can now be written in its squared form for simplicity:

$$(x+3)^2 = 9$$

Step 3: Now we can just solve as before. First we take the square root of both sides:

$$x + 3 = \pm\sqrt{9}$$

Step 4: Subtract 3 from both sides to leave **x** isolated on the LHS:

$$x = -3 \pm \sqrt{9}$$

This results in the two solutions for **x** shown below:

$$x = -3 + 3 = 0$$

and

$$x = -3 - 3 = -6$$

We have successfully found the solutions to the equation $2x^2 + 12x = 0$ by first completing the square and then solving as usual.

The technique of completing the square can be reliably used to solve any quadratic. Let us look at one more example of completing the square before moving on.

Example 7: $4x^2 + 12x - 100 = 0$ (Solving a Quadratic Equation that is not a square.)

Step 1: In this example, our equation is a trinomial to begin with, but not a perfect square. In a perfect square trinomial we need the 3rd term to be something very specific (e^2), so we really want to lose our current 3rd term on the LHS so that we can put something else there that completes the square. So let us first add 100 to both sides of the equation, which will remove the 3rd term (-100) from the LHS and add 100 to the RHS. After this step, our equation now looks as follows:

$$4x^2 + 12x = 100$$

Step 2: Our next job is to remove the coefficient of the squared term ($\mathbf{x^2}$) so that it matches its like term in the identity. If we divide every term by 4, this will eliminate the coefficient of the squared term. More correctly, it reduces it to an implied coefficient of 1, as shown below. Notice how dividing $\mathbf{4x^2}$ by 4 leaves us with $\mathbf{x^2}$, exactly as we want:

$$x^2 + \frac{12}{4}x = \frac{100}{4}$$

If we carry out the remaining divisions indicated above we can see that 12x/4 = 3x and 100/4 = 25, reducing the equation to the following:

$$x^2 + 3x = 25$$

Step 3: Now we have the LHS reduced to two terms with no leading coefficient accompanying the squared term. So we find ourselves with an equation in the same form as the previous example. We know from the identity that the coefficient of **o** in the middle term is **2e**, and we can see that as **x = o** in our equation, its coefficient **3** must be equal to **2e**. Therefore we have correctly identified that **e = 3/2 = 1.5**. We know that we can now complete the square by adding e^2 to both sides of the equation. Since **e = 1.5** and $\mathbf{e^2 = 2.25}$, we simply add **2.25** to both the LHS and the RHS to complete the square:

$$x^2 + 3x + 2.25 = 27.25$$

Step 4: Now that we have a PST on the LHS and we know that **o = x** and **e = 1.5**, we can write the LHS in $(o + e)^2$ form, as shown below:

$$(x + 1.5)^2 = 27.25$$

Step 5: We take the square roots of both sides of the equation, leaving us with x + 1.5 on the LHS and the positive and negative square roots of 27.25 on the RHS (which we will leave under the square root sign for now).

$$x + 1.5 = \pm\sqrt{27.25}$$

Step 6: Finally we can isolate **x** by subtracting 1.5 from the both sides of the equation, introducing –1.5 on the RHS.

$$x = -1.5 \pm \sqrt{27.25}$$

We now have the two values of **x** shown below. The first shows the calculation using the positive root of 27.27 (5.2201) .

$$x = -1.5 + 5.2201 = 3.7202$$

The second value of x is determined using the negative root of 27.25 (-5.2201)

$$x = -1.5 - 5.2201 = -6.7202$$

Therefore, the solutions are **x = 3.7202** and **x = –6.7202**.

Over the past few sections we have learned how to solve both simple and complex quadratic equations using a method known as completing the square. When completing the square we first mold the LHS into a perfect square trinomial. But early on, we discussed how to take a quadratic expression in any form an manipulate it into the standard form $at^2 + bt + c = 0$.

Since we learned how to solve standard and non-standard form quadratic equations, you may be wondering why we even bothered with the discussion of manipulating an equation into the standard form. We certainly do not need it in that form to solve it as we have shown. But while that is technically true, we do need to be able to solve quadratic equations in a consistent manner using a single method. This is especially true for us as programmers as we will need to write code that solves quadratic equations. We certainly do not want to have lots of conditionals testing the form of the quadratic equation so that a way to solve it can be determined.

As it happens, any quadratic equation that is in the standard form can be solved using a simple calculation called the quadratic formula. Thus if we want to write a function that can solve any quadratic equation with a minimum of code, we can just insist the equation passed into the function that needs to be solved is in the standard form. We can then use the quadratic formula to find the roots of the equation every time (assuming they exist). Whether the quadratic equation is being solved to determine the intersection between a ray and a sphere, a ray and a cylinder, or even for calculating cost analysis, provided it has been manipulated into the standard form, our code can always solve it using this simple technique. Naturally then, our next and final discussion on solving quadratic equations will be on the quadratic formula, since it is the method we will be using for solving our standard form quadratic equations.

12.6.3 The Quadratic Formula

It was determined quite a long time ago that the quadratic formula could be used to solve any quadratic equation in the standard form. Indeed, the quadratic formula is so useful that it has become required learning in high schools around the world. It really is an essential tool in the mathematicians' toolkit and can be used to solve quadratics very easily.

Provided a quadratic equation is in the form $at^2 + bt + c = 0$, we can solve for *t* using the quadratic formula, which is shown below.

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The Quadratic Formula

It is well worth remembering this formula so that you can recall it at will. It will come in extremely handy down the road and will be the method we use when solving for intersections between a ray and a quadric surface (sphere or cylinder). Remember that the $\pm$ symbol essentially says that there is a positive and negative root of **b^2 - 4ac.** This indicates that as long **b^2 - 4ac** is not zero, there will be two real solutions for *t*. If we call the two roots *t1* and *t2*, we see the two solutions for *t* below.

$$t1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$t2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

The section of the formula under the square root sign $b^2 - 4ac$ has special properties which allow us to determine how many *real* roots (if any) can be found prior to performing the full calculation. For this reason, it is called the *discriminant* of the polynomial. We can see for example, that if the discriminant is zero, then both *t1* and *t2* will be identical and thus we might say that one real root exists.

We can also see that if the discriminant is a negative value, then the equation has no real roots. This lets us avoid applying the full quadratic formula logic since we know that we cannot take the square root of a negative number and expect a real result. This suggests to us that we should calculate the discriminant first and determine if it is a negative number. If so, no real roots exist.

Below we see how we might implement a new function called SolveRoots. It is passed the **a**, **b**, and **c** coefficients of a quadratic in standard form and will return the real roots of the equation, if they exist, in the output variables t1 and t2.

```
bool CCollision::SolveRoots( float a, float b, float c, float& t1, float &t2 )
{
    float d, one_over_two_a;

    // Calculate Discriminant first
    d = b*b - 4*a*c;

    // No discriminant is negative return false
    // as no real roots exist
    if (d < 0.0f) return false;

    // Calculate square root of b^2-4ac ( discriminant )
    d = sqrtf( d );

    // Calculate quadratic formula denominator ( 2a )
    one_over_two_a = 1.0f / (2.0f * a);

    // Calculate the two possible roots using quadratic formula.
    // They be duplicate is d = 0
    t1 = (-b - d) * one_over_two_a;
    t2 = (-b + d) * one_over_two_a;

    // Solution found
    return true;
}
```

These few lines of code are pretty much all there is to solving a quadratic equation in standard form using the quadratic formula. The actual function that solves the quadratic equations in our CCollision class is in a method called SolveCollision which is almost identical to this. The only difference is that the SolveCollision function will only return the smallest positive root. This is because the quadratic equation passed in (as we will see later) will represent the intersection of a ray with a sphere or cylinder. As negative *t* values will represent the distance to the surface behind the ray origin we are not interested in returning them. Additionally, since we wish to find the first point of intersection between the ray and the quadric surface (there may be two intersections since a ray can pass through a sphere's surface twice), we are only interested in the nearest intersection. Therefore, we are interested in the **smallest positive** *t* value.

Below we see the actual code to our CCollision::SolveCollision function.

```
bool CCollision::SolveCollision( float a, float b, float c, float& t )
{
    float d, one_over_two_a, t0, t1, temp;

    // Basic equation solving
    d = b*b - 4*a*c;

    // No root if d < 0
    if (d < 0.0f) return false;

    // Setup for calculation
    d = sqrtf( d );
    one_over_two_a = 1.0f / (2.0f * a);

    // Calculate the two possible roots
    t0 = (-b - d) * one_over_two_a;
    t1 = (-b + d) * one_over_two_a;

    // Order the results
    if (t1 < t0) { temp = t0; t0 = t1; t1 = temp; }

    // Fail if both results are negative
    if (t1 < 0.0f) return false;

    // Return the first positive root
    if (t0 < 0.0f) t = t1; else t = t0;

    // Solution found
    return true;

}
```

Notice that after we have found the two solutions, we store them in local variables t0 and t1. We then order them (possibly involving a swap) so that t0 always holds the smallest *t* value and t1 holds the larger one. We then test to see if t1 is smaller than zero. If it is then both *t* values are negative (because t1 is the largest value) and the intersections between the ray and the quadric surface happen behind the ray origin and are of no interest to us. Finally, we test if t0 (which now contains the smallest *t* value) is negative. If it is, then we return t1 which will be the smallest positive root. If t0 is not negative, then it must contain the smallest positive root, since we ordered them this way a few lines before. If this is the case, we return the value of t0.

Of course, at this point you are not supposed to understand how this function gets called and why we need to call it. This will all be discussed in a moment. However, hopefully you can see that this function is essentially doing nothing more than being passed the coefficients of a quadratic equation in standard form and then solving for the unknown *t*. If you follow the lines of code, you should see how we are performing the steps exactly as dictated by the quadratic formula. Regardless of whether we wish to intersect a ray with a sphere, a cylinder, or any other quadric surface, provided we can substitute the ray equation into the equation of the surface (discussed in a moment) and then manipulate that equation into the standard form, we can use this function to find the *t* values (i.e., the points of intersection parametrically along the ray).

Thus far you have taken for granted that the quadratic formula works but we have not discussed why it works and how it was arrived at in the first place. Therefore, let us see what happens if we try to solve a generic quadratic equation in standard form. You will find that we end up with the quadratic formula. This means you will understand not only what the quadratic formula is, but also why it works.

Step 1: We first determine that we wish to solve a quadratic equation in the standard form. Our objective is to isolate *t* on the LHS thus finding solutions for it. We will do this by manipulating the LHS into a perfect square in the form $\mathbf{o^2 + 2eo + e^2}$ which can then be written as $\mathbf{(o+e)^2}$ and easily solved. So let us start with the standard form quadratic equation template.

$$at^2 + bt + c = 0$$

Step 2: The first simple thing we can do in our quest to remove everything from the LHS (other than **t**) is to subtract **c** from both sides. This will eliminate the third term from the LHS and introduce **–c** on the RHS as shown below.

$$at^2 + bt = -c$$

Step 3: To make our perfect square trinomial, we know that the squared term $\mathbf{t^2}$ must have an implied coefficient of 1. Therefore, just as we have done many times before, we will divide the rest of the terms of the equation by **a**. This will set the coefficient of the squared term to 1 and leave just $\mathbf{t^2}$.

$$\frac{a}{a}t^2 + \frac{b}{a}t = \frac{-c}{a}$$

We know that a/a is just 1 and cancels itself out, so our equation now looks like what we see next. It is shown along side the PST whose form we are trying to replicate.

$$t^2 + \frac{b}{a}t = \frac{-c}{a}$$

$$o^2 + 2eo + e^2 = 0$$

Step 4: Looking at our equation we can see that **t** is equivalent to **o** in the PST. We can also see that the coefficient of **t** in the middle term of our equation is **b/a** which is equal to **2e** (the coefficient of **o** in the PST's middle term). Therefore, we have determined that **2e = b/a** in our equation and thus, **e** must equal **b/a** divided by 2.

When we multiply a fraction by a single value, we essentially treat that single value as a fraction with a denominator of 1. Therefore, multiplying **b/a** by 2 would look like this.

$$\frac{b}{a} x \frac{2}{1} = \frac{b*2}{a*1} = \frac{2b}{a}$$

As you can see, we multiply the two fractions by multiplying the numerators and denominators separately. When we wish to divide a fraction by a real number we essentially do the same thing. That is, we add a denominator of 1 to the single value to get it into fraction form. However, because we wish to divide one fraction by another instead of multiply, we still perform a multiply but will swap the numerator and the denominator of the second fraction (i.e., multiply by the *reciprocal*).

Therefore, since we know that **2e = b/a** in our equation, we can find the value of **e** by multiplying the fraction **b/a** with the fraction ½ as shown below.

$$e = \frac{b}{a} x \frac{1}{2} = \frac{b}{2a}$$

Step 5: Now that we know the value of **e**, we can add $\mathbf{e^2}$ to both sides of the equation to form a perfect square trinomial on the LHS which is ready to be written in its squared form. As we know that **e=b/2a**, $\mathbf{e^2}$ must be equal to multiplying the fractions (**b/2a**)*(**b/2a**) as shown below.

$$e^2 = \frac{b}{2a} x \frac{b}{2a} = \frac{b^2}{4a^2}$$

So we must add this result to both the LHS and the RHS of our equation to complete the square. Our equation now looks like this:

$$t^2 + \frac{b}{a}t + \frac{b^2}{4a^2} = -\frac{c}{a} + \frac{b^2}{4a^2}$$

Step 6: As we now know that we have a perfect square trinomial on the LHS and that **o = t** and **e = b/2a**, we can write the LHS of the equation in its squared form.

$$\left(t + \frac{b}{2a}\right)^2 = \frac{-c}{a} + \frac{b^2}{4a^2}$$

Step 7: At this point the RHS could do with a little cleaning up. It is currently represented as the addition of two fractions, which is not very nice. But we could merge these two fractions together if they both had the same denominator. For example, if the left fraction on the RHS had a denominator of $4a^2$ also, we could represent the RHS simply as

$$\frac{-c+b^2}{4a^2}$$

Of course, we cannot do that because the left fraction does not have a denominator of $4a^2$. However, we could multiply the left fraction (both the numerator and the denominator) so that it is scaled up to a fraction of the same proportion but with a denominator of $4a^2$. Many of you may remember this from high school as finding the lowest common denominator. The goal is to have all fractions use the same denominator to simplify our calculations.

What can we multiply the first fraction (**–c/a**) by such that its numerator and denominator stay proportional to one another and yet we wind up with a fraction whose denominator has $4a^2$? That is simple enough -- we multiply **–c/a** by the fraction **4a/4a** since multiplying the first fraction's denominator (**a)** by **4a** will scale it to **$4a^2$** as shown below.

$$\frac{-c}{a} * \frac{4a}{4a^2} = \frac{-4ac}{4a^2}$$

As you can see, multiplying the numerator **–c** by **4a** results in a numerator of **–4ac**. Multiplying the denominator **a** by **4a** results in the denominator we were after: **$4a^2$**. This now matches the denominator of the other fraction on the RHS.

$$\left(t+\frac{b}{2a}\right)^2 = \frac{-4ac}{4a^2} + \frac{b^2}{4a^2}$$

Since both the fractions on the RHS now have the same denominator, we can add the two fractions together to get a single result on the RHS:

$$\left(t+\frac{b}{2a}\right)^2 = \frac{-4ac+b^2}{4a^2}$$

What we will do now is rearrange the numerator of the RHS (we generally prefer to have the terms sorted by power going from left to right) to give:

$$\left(t+\frac{b}{2a}\right)^2 = \frac{b^2-4ac}{4a^2}$$

Those of you paying close attention will have noticed that the numerator on the RHS is actually the discriminant we discussed earlier.

Step 8: We currently have our equation in squared form. As we have done many times before, we now wish to take the square root of both sides of the equation. This will remove the squared brackets on the LHS leaving us with just **t + b/2a**, making **t** easier to isolate.

$$\sqrt{\left(t+\frac{b}{2a}\right)^2}=\sqrt{\frac{b^2-4ac}{4a^2}}$$

The square root of the LHS is simple. It just removes the squared brackets like so.

$$t+\frac{b}{2a}=\sqrt{\frac{b^2-4ac}{4a^2}}$$

Taking the square root of the RHS is easy also. We simply take the square root of the denominator and the numerator separately.

$$t+\frac{b}{2a}=\frac{\sqrt{b^2-4ac}}{\sqrt{4a^2}}$$

Looking at the numerator of the RHS first; the square root of $\mathbf{b^2-4ac}$ is something we just do not know since it cannot be reduced any further. So we will just leave the numerator as is with the square root sign still in place. However, the denominator is no problem at all. We know that:

$$2a*2a=4a^2$$

Thus, the square root of $4a^2$ is 2a. After we have taken the square roots of both sides, our equation now looks like this.

$$t+\frac{b}{2a}=\frac{\pm\sqrt{b^2-4ac}}{2a}$$

Step 10: We nearly have **t** isolated on the LHS. All that is left to do now is subtract **b/2a** from both sides of the equation:

$$t = \frac{\pm\sqrt{b^2 - 4ac}}{2a} - \frac{b}{2a}$$

Because both fractions on the RHS share a common denominator, we can add them and rearrange to look a little nicer. The result is the quadratic formula:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

This wraps up our review of quadratic equations. As explained at the beginning of this section, the discussion has been completely abstracted from the topic of collision detection and as such, you may be wondering what any of this has to do with finding intersections with triangles. In the next section, you will finally get the answers to your questions and see that everything you have learned here will be put to good use.

12.7 The SphereIntersectTriangle Function: Part II

Earlier we discussed the progress we had made with respect to writing a function that would correctly detect the intersection point between a swept sphere and a polygon. As discussed, we currently have only implemented the first and most common case – testing against the plane. If the sphere did not intersect the plane of the triangle, the function can immediately return false. If an intersection with the plane has occurred then we tested the intersection point (on the plane) to see whether it was inside the polygon/triangle edges. If it was, then our task was complete. We have our intersection point on the shifted plane (the potential new sphere center position) and we have our intersection normal (the normal of the polygon which we can use for sliding). However, if we have intersected the plane but the intersection point is not in the triangle interior, then we must test for other cases. For starters, we must test if the swept sphere has collided with one of the three triangle edges (Figure 12.31).

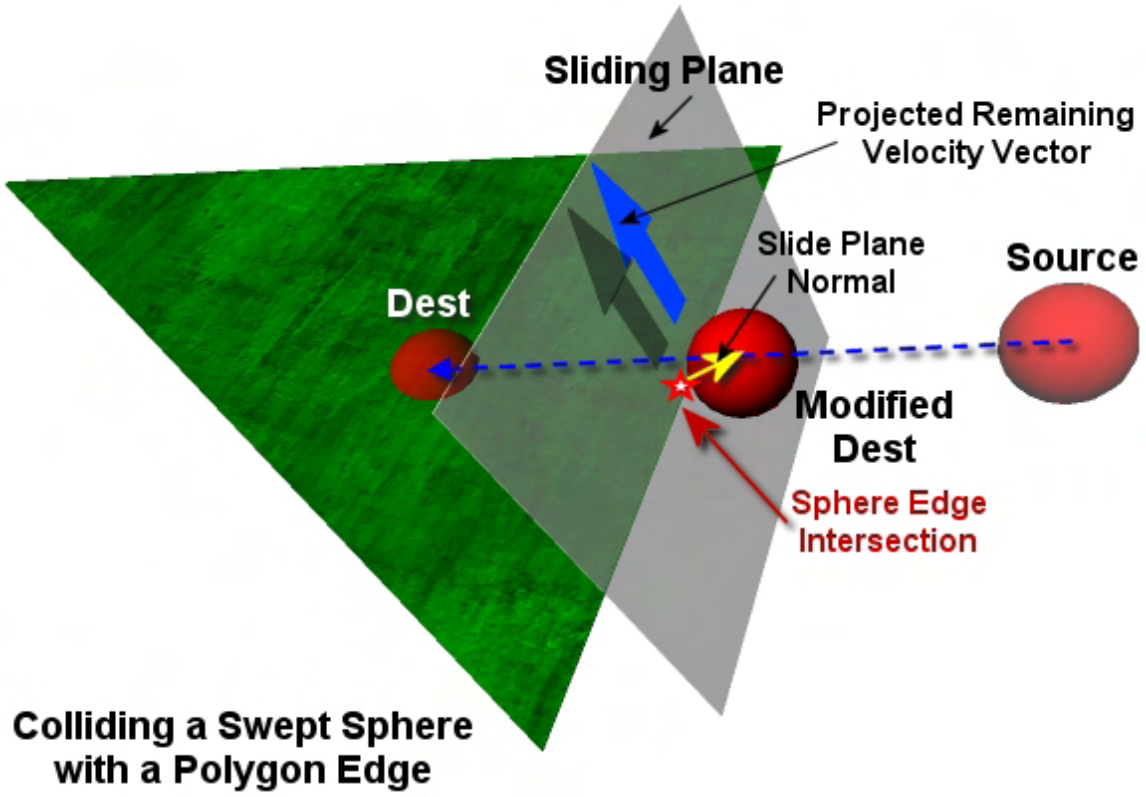


Figure 12.31

Figure 12.31 shows an intersection test between our swept sphere and a single edge. We will need to perform this test for each edge to make sure we catch all collision cases and return the closest intersection.

In Figure 12.31, **Source** and **Dest** describe the initial center position of the sphere and the desired destination of the sphere after the movement update. The blue dashed line connecting them describes the velocity vector we wish to move the sphere along to make that so. **Modified Dest** is the new center position we need to calculate which will position the sphere such that its surface is touching (but not intersecting) the triangle edge.

Once we have found this new position, we also have to return an intersection normal for later use as our slide plane normal. As Figure 12.31 shows, this will be calculated by creating a vector from the intersection point on the edge to the sphere center position and then normalizing the result. This is different from the normal returned when the sphere collides with the interior of a polygon. In the other case, the triangle normal is returned. But when we collide with an edge, we wish to slide around that edge smoothly. You should be able to see that the angle of approach made by the sphere with the intersected edge influences the orientation of the slide plane generated. If the sphere collides with the edge right in the center of its surface a very steep plane will be produced. With such a steep slide plane, any remaining velocity vector will be scaled down quite a bit when projected onto the slide plane (in the response phase) and only a little slide will happen. This is like hitting the edge head on. If however, the intersection occurs via a more shallow angle of approach (almost as if we are just clipping the edge

accidentally as we walk by it), the plane will not be as steep and the edge will be easily slid around without losing to much velocity during the velocity vector projection process in the response step.

While we do not yet know how to write a function that determines the intersection between a swept sphere and a triangle edge, for now we will stipulate that this test will be performed in a method called CCollision::SphereIntersectLineSegement. This will be the method that will be called for each edge of the triangle to test for intersection between the sphere and the line segment formed by the vertices of each edge. While we will discuss how to implement this function in a moment, we now have enough information to understand the final version of our CCollision::SphereIntersectTriangle function. The final code is shown below. The only difference from the version we saw earlier will be the three function calls that have been added to the bottom of the function (highlighted in bold).

```
bool CCollision::SphereIntersectTriangle(  const D3DXVECTOR3& Center,
                                           float Radius,
                                           const D3DXVECTOR3& Velocity,
                                           const D3DXVECTOR3& v1,
                                           const D3DXVECTOR3& v2,
                                           const D3DXVECTOR3& v3,
                                           const D3DXVECTOR3& TriNormal,
                                           float& tMax,
                                           D3DXVECTOR3& CollisionNormal )
{
    float       t = tMax;
    D3DXVECTOR3 CollisionCenter;
    bool        bCollided = false;

    // Find the time of collision with the triangle's plane.
    if ( !SphereIntersectPlane( Center, Radius, Velocity, TriNormal, v1, t ))
        return false;

    // Calculate the sphere's center at the point of collision with the plane
    if ( t < 0 )
        CollisionCenter = Center + (TriNormal * -t);
    else
        CollisionCenter = Center + (Velocity * t);

    // If this point is within the bounds of the triangle,
    // we have found the collision
    if ( PointInTriangle( CollisionCenter, v1, v2, v3, TriNormal ) )
    {
        // Collision normal is just the triangle normal
        CollisionNormal = TriNormal;
        tMax            = t;

        // Intersecting!
        return true;

    } // End if point within triangle interior

    // Otherwise we need to test each edge
    bCollided |= SphereIntersectLineSegment( Center, Radius, Velocity,
                                             v1, v2, tMax, CollisionNormal );

    bCollided |= SphereIntersectLineSegment( Center, Radius, Velocity,
```

```
                                          v2, v3, tMax, CollisionNormal );

    bCollided |= SphereIntersectLineSegment( Center, Radius, Velocity, v3, v1,
                                             tMax, CollisionNormal );

    return bCollided;
}
```

As you can see, if the sphere is intersecting the plane of the triangle and the intersection point is inside the triangle we return true. However, if the intersection point is not within the interior of the triangle we perform an intersection test between the swept sphere and each edge. We pass into the SphereIntersectLineSegment function the eSpace position of the center of the ellipsoid, the radius vector of the ellipsoid, and the eSpace velocity vector describing the direction and distance we wish to move the sphere. We also pass in the two vertices that form the edge we are testing. Obviously a triangle has three edges, so the function is called three times (once to test each edge). Each time a different vertex pair is used to describe the edge.

We also pass (by reference) the tMax variable that was itself passed in as a parameter to the function. Remember from our earlier discussions that this will always hold the smallest intersection distance (*t* value) that we have found so far. We are only interested in finding intersections that are closer than this value since if the intersection is further away, it means we have previously found another triangle that is closer to our sphere. Remember, the SphereIntersectTriangle function is called for every triangle in the collision database and the same tMax variable is overwritten as soon as we find an intersection that is closer than one we previously stored.

As the final parameter, we pass (by reference) CollisionNormal, which itself is a vector that was passed by reference into the function. If an intersection is found with the edge that is closer than any found so far, the function will store the collision normal in this variable. When the SphereIntersectLineSegment function returns, if an intersection has been determined to be closer than any found with previously tested triangles, the calling function will have access to both the intersection *t* value and the collision normal.

So the question that begs to be asked is, how do we detect an intersection between a moving sphere (swept sphere) and a polygon edge? The answer can be found in the next section.

12.7.1 Swept Sphere / Triangle Edge Intersection

Performing an intersection test between a swept sphere and a triangle edge sounds like it might be quite complicated. When we tested the swept sphere against a plane, we were able to reduce the entire test to a ray / plane test simply by shifting the plane along its normal by the radius of the sphere. The point at which the ray intersected the shifted plane described the exact center position of the sphere at the time the surface of the sphere will make contact with the actual plane. Certainly it would be nice if a similar technique could be employed to reduce the swept sphere / edge test to that of an intersection test between a ray and some other primitive type. As it turns out, there is indeed a way to do this.

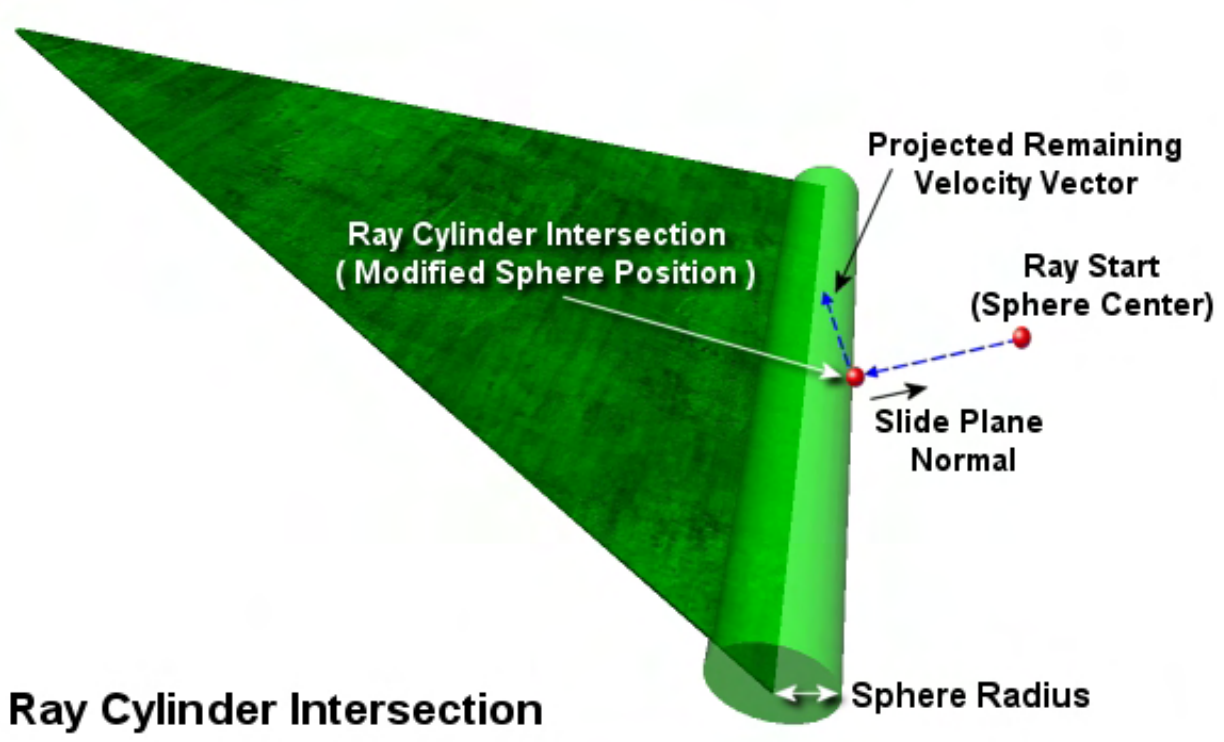


Figure 12.32

In the sphere / plane intersection test we were able to reduce the swept sphere to a ray by shifting the plane along the normal by the sphere radius. Imagine now, that we were to take the edge of the triangle that we currently want to test and expanded it outwards in all directions by the radius of the sphere. That is, we would give the edge a thickness equal to twice the sphere's radius. If we could do this, we would have essentially grown the edge into a cylinder, where any point on the surface of the cylinder will be located at a distance of sphere radius from its central spine.

As Figure 12.32 illustrates, if the edge becomes a cylinder, the swept sphere can once again be treated as a ray, where the original eSpace center position of the sphere becomes the ray origin (Ray Start) and the eSpace velocity vector becomes the ray's delta vector. If we find the intersection point along the ray with the surface of the cylinder, it will be at a distance of sphere radius from the actual edge and will thus describe the exact location where the center of the sphere should be positioned at the time the surface of the sphere is touching the actual edge (the time of intersection). Remember, the true edge is the central spine of the cylinder which is at a distance of radius units from any point on the surface of the cylinder. All we have to do is implement a function that will find the *t* value along a ray which intersects the surface of a cylinder. This will provide us with the new position for the center of the sphere at the time of intersection between the sphere's surface and the triangle edge.

Once we have this intersection point on the surface of the cylinder, we can also calculate the slide plane normal by generating a vector from the intersection point on the actual edge to the new sphere center we just calculated (and normalizing the result).

The equation for a cylinder is.

$$\left\|(P-V1)\times E\right\|^2 = Radius^2$$

Let us analyze this formula. **P** is a point on the surface of the cylinder and **V1** is the first vertex in the edge that forms the cylinder's central spine (axis). This is the point at the base of the cylinder. **E** is the unit length direction vector of the cylinder's central spine. Therefore, if the edge of the triangle is given by vertices **V1** and **V2**, **E** is created by normalizing the vector (**V2 - V1**) as shown in Figure 12.33.

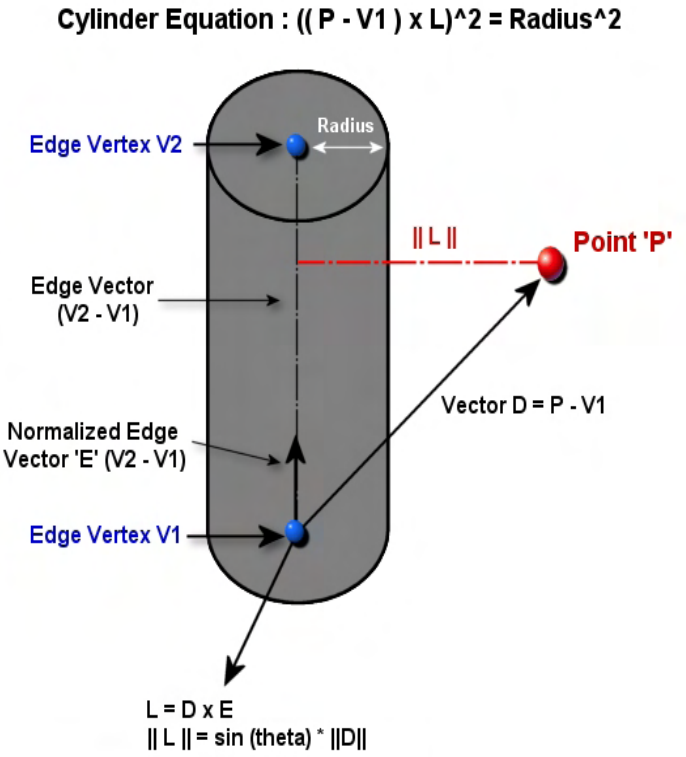


Figure 12.33

The cylinder equation says that if the squared length of a vector, (generated by crossing a vector from **V1** to the point **P** with the unit length direction vector of the cylinder's central axis **E**) is equal to the squared radius of the cylinder, the point **P** is on the surface of the cylinder. In Figure 12.33, the point **P** is clearly not on the surface of the cylinder, but what is the relationship between the squared length of the vector we have created from that cross product (**(P-V1)xE**) on the LHS of the equation and the radius of the cylinder?

In this example, **P** in the cylinder equation is **Point P** in the diagram. We can see that it is not on the surface of the cylinder, but this was done intentionally to clarify the relationships we are about to discuss. We can also see that if we create a vector from **V1** at the bottom of the cylinder (first vertex in the edge) to the point **P**, we have vector **D** in the diagram. If we look at unit vector **E**, we can imagine how this forms the adjacent side of a triangle where the vector **D** forms the hypotenuse. What we need to know is the distance from **Point P** to the edge (red dashed horizontal line in Figure 12.33). To calculate this distance we start by performing the cross product between vectors **D** and **E**. We know that when we perform the cross product between two vectors we get a vector returned that is perpendicular to the two input vectors. This is vector **L** in the diagram (assumed to be pointing out of the page). The direction of this vector appears to have nothing to do with what we are trying to accomplish, and indeed it does not. However, our interest in is the length of this vector, not its direction. Why? Because the length of the vector returned from a cross product given as:

$$\left\|D\times E\right\| = \left\|D\right\|\left\|E\right\|\sin\alpha$$

The length of the vector returned from crossing **D** and **E** is equal to the sine of the angle between them multiplied by the length of vector **E** and the length of vector **D**. In our cylinder equation, **E** is a unit length vector and thus this equation simplifies to:

$$\|D \times E\| = \|D\| \sin \alpha$$

So when we cross **D** and **E**, we are returned a vector whose length is equal to the sign of the angle between the two input vectors scaled by the length of vector **D** (the hypotenuse).

Now, if you refer back to Figure 12.33 and envisage **D** as the hypotenuse and **E** as the adjacent of a right angled triangle, we can see that what we actually wish to determine is the length of the opposite side of that triangle. The opposite side is the line from the point **P** to the adjacent side formed by the central spine of the cylinder. How do we get the length of the opposite side of a right angled triangle?

The helpful acronym **SOH CAH TOA** tells us that we can find the sine of the angle by dividing the length of the opposite side by the length of the hypotenuse.

$$\sin \alpha = \frac{Opposite}{Hypotenuse}$$

Therefore, in order to find the length of the opposite side, we simply multiply each side of this equation by the length of the hypotenuse leaving the opposite side isolated on the RHS.

$$\sin \alpha * Hypotenuse = Opposite$$

or more nicely arranged:

$$Opposite = Hypotenuse * \sin \alpha$$

So we know that we can find the length of the opposite side of a triangle by multiplying the length of the hypotenuse by the sine of the angle. Does that help us? Yes it does, because that is exactly what the cross product just gave us. Remembering once again that vector **D = (P – V1)** is the hypotenuse of our triangle and vector **E** is unit length, the cross product of **DxE** returns a vector **L** with the following length.

$$\|L\| = \|D \times E\|$$

The above states that our vector **L** is the result of crossing vectors **D** and **E**. The length of vector **L** (||**L**||) is then obviously the length of the vector returned from crossing **D** and **E**. Since vector **D** is generated by subtracting vector **V1** from point **P**, using substitution for **D**, this can also be written as:

$$= \|(P - V1) \times E\|$$

The result of the cross product is a vector whose length is equal to the sine of the angle between them multiplied by the lengths of each input vector. Since input vector **E** is unit length, the result is simply the sine of the angle scaled by the length of the non-unit vector **D** (i.e., **P - V1**).

$$= \|(P - V1)\| \sin \alpha$$

Since vector **P-V1** forms the hypotenuse of our triangle, the length of this vector is equal to the length of the hypotenuse, so we can write this as:

$$= \|Hypotenuse\| \sin \alpha$$

As we just discovered, scaling the length of the hypotenuse of a right angled triangle by the sine of its angle gives us the length of the opposite side,

$$= \|Opposite\|$$

And there we have it. As **D** is the hypotenuse and **E** is unit length, the length of the resulting vector **L** is equal to the length of vector **D** multiplied by the sine of the angle between **D** and **E**. This is the length of the hypotenuse multiplied by the sine of angle, which trigonometry tells us is the length of the opposite side. This is the length of the red dashed horizontal line in the diagram and is the distance from point **P** to the cylinder's central spine (i.e., the triangle edge).

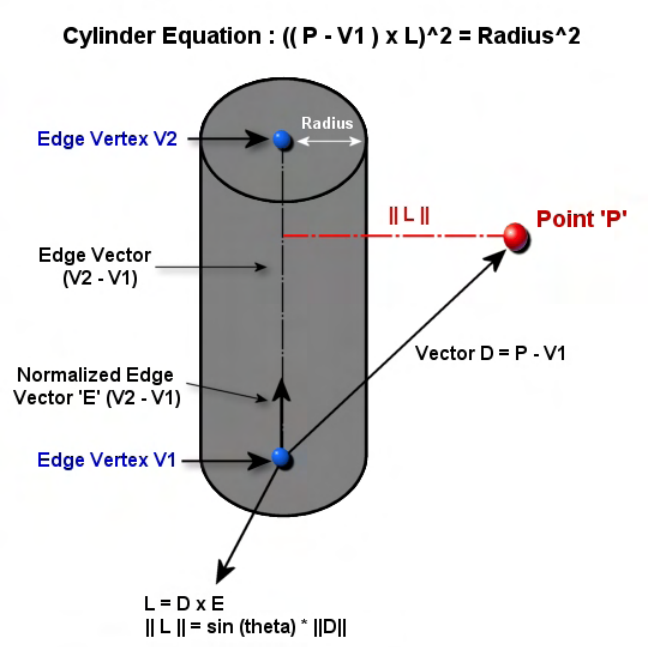


Figure 12.34

In Figure 12.34 we can now see exactly why we compare the squared length of vector **L** against the squared radius. The length of vector **L** describes the distance from the point **P** to the edge (the central axis). Since the cylinder's surface is a distance of Radius units from its central axis, if the squared distance from point **P** to the edge (||**L**||) is equal to the Radius squared, the point must be on the surface of the cylinder. In Figure 12.34 we can see that the length of vector **L** would be much greater than the radius of the cylinder and therefore the point **P** is not considered to be on the surface of the cylinder.

Of course, if we were only interested in determining whether or not a point is on the surface of the cylinder, we would be all set. However, we wish to find a ***t*** value along a ray which describes a point on the surface of the cylinder (i.e., the *t* value at which the ray intersects the cylinder). Just as we did when intersecting a ray with a plane, we must substitute the equation of our ray into the cylinder equation in place of point **P**.

If the cylinder equation is:

$$\|(P - v1) \times E\|^2 = Radius^2$$

and the equation of our ray is defined by an origin **O**, a delta vector **V**, and a parametric value ***t*** describing a point along that ray:

$$O + tV$$

Substituting our ray into the cylinder equation in place of **P** gives us:

$$\|(O + tV - V1) \times E\|^2 = Radius^2$$

Because we are dealing with vectors from an algebraic point of view, we can treat the double bars || as brackets, since we know that they mean the length of a vector. Because this length is squared in the equation, we also know that we can get the squared length of a vector by dotting it with itself (a vector multiply). Therefore, let us re-write our equation in bracket form. Many of you may consider this approach a little relaxed, but we find it helpful.

$$((O + tV - V1) \times E)^2 = Radius^2$$

This is not a very attractive equation at the moment since we want to isolate ***t*** as much as possible. Well, we know that subtracting vector **V1** from the vector (**O + tV**) is equivalent to subtracting vector **V1** from vector **O** and then adding vector **tV** afterwards, so we can swap them around.

$$((O - V1 + tV) \times E)^2 = Radius^2$$

We also know, that because the vector (**O-V1**) and the vector **tV** are inside brackets that are crossed with vector **E**, this is equivalent to removing those inner brackets and crossing the vector (**O-V1**) and **tV** with **E** separately. As you can hopefully see, we are slowly breaking this equation down into smaller more manageable and components.

$$((O - V1) \times E + (tV \times E))^2 = Radius^2$$

We also know that *t* is a scalar not a vector. Scaling vector **V** by *t* and then crossing with **E** is equivalent to crossing **V** and **E** and then scaling the result. Therefore, we can move *t* outside the brackets, isolating it even more.

$$((O - v1) \times E + (V \times E)t)^2 = Radius^2$$

This is all looking good so far. However, everything on the LHS is still in squared form. Thus, everything on the LHS can be removed from the squared brackets by multiplying the LHS with itself as we discussed earlier (refer back to multiplying an expression in squared brackets). Since the expression in both brackets is a vector, and the vector form of a multiply is a dot product, we should use the dot symbol as shown below when we multiply the expression by itself:

$$((O - v1) \times E + (V \times E)t) \bullet ((O - v1) \times E + (V \times E)t) = Radius^2$$

This is starting to look a little more complex, but that is no problem. We just have a set of bracketed terms that we have to perform **FOIL** on. For the time being, we will just concentrate on the LHS of the equation.

The **First** term in each bracket is $(O - v1) \times E$. Therefore, we are multiplying vector $(O - v1) \times E$ by itself. This gives us a new first term of:

$$((O - v1) \times E) \bullet ((O - v1) \times E) \; = \; (O - v1) \times E)^2$$

The **Outer** term of the first bracket is $(O - v1) \times E$ whilst the outer term of the second bracket is $(V \times E)t$. Both of these are vectors and therefore, our second new term is calculated as:

$$((O - v1) \times E) \bullet ((V \times E)t)$$

The **Inner** terms of the brackets are $(V \times E)t$ and $(O - v1) \times E$ for the first and second brackets respectively, giving us an identical term to the last one if we swap them around (which we can do without altering its evaluation):

$$((O - v1) \times E) \bullet ((V \times E)t)$$

Finally, we multiply the last terms of each bracket, which are the same for both $(V \times E)t$:

$$(V \times E)t \bullet (V \times E)t \quad = \quad (V \times E)^2 t^2$$

Let us now add the four results together and see our new expanded equation. We have put each FOIL result on its own line for clarity at this stage, but this is all just the LHS of our equation.

$$\begin{aligned}
&((O - v1) \times E)^2 + \\
&((O - v1) \times E) \bullet (V \times E)t + \\
&((O - v1) \times E) \bullet (V \times E)t + \\
&(V \times E)^2 t^2
\end{aligned}$$

One thing is immediately obvious. Since *t* is the unknown we wish to find and it is raised to a power of two in one of the expressions, we have ourselves a quadratic that needs solving. We know how to solve quadratic equations using the quadratic formula, but we need our equation in the standard form.

$$at^2 + bt + c = 0$$

Looking at our equation and searching for the squared term, we can immediately identify the value of our **a** coefficient. It is the expression that shares the term with $\mathbf{t^2}$, $(V \times E)^2$. Therefore, let us move that entire expression to the front of the equation so that it resembles its position in the standard form.

$$(V \times E)^2 t^2 +$$
$$((O - v1) \times E)^2 +$$
$$((O - v1) \times E) \bullet (V \times E)t +$$
$$((O - v1) \times E) \bullet (V \times E)t$$

We have the squared term exactly where it should be now and we can quickly see that

$$a = (V \times E)^2$$

In other words, **a** should be set to the squared length of the vector produced by crossing the velocity vector (the ray delta vector) with the unit length direction vector of the edge (the cylinder's central spine).

We have to find the **a**, **b**, and **c** coefficients in order to solve our quadratic, and we now have one of them. So let us now work on finding the **b** coefficient.

We know from studying the standard form that **b** must be calculated from expressions that accompany *t* (not t^2). There are actually two terms in our above equation that contain *t* in its non-squared form – currently, the third and fourth terms (which are actually identical terms). Therefore, we know that we must be able to construct **b** by isolating *t* from these terms. The two terms of our equation that contain *t* in our current equation are shown below.

$$((O - v1) \times E) \bullet (V \times E)t + ((O - v1) \times E) \bullet (V \times E)t$$

Since these two expressions are the same, we are essentially adding $((O - v1) \times E) \bullet (V \times E)t$ twice to the LHS of our equation. In each term we are taking the dot product of two vectors **(O-v1)xE** and **VxE** and scaling the result by *t*. Since the result of the dot product in each of the above terms is scaled by *t*, we can scale the sum of the combined dot products by *t* in a single step instead of separately in each term. What that means (in generic terms) is, instead of doing something like **A.Bt+A.Bt,** we can instead write **2(A.B)t**. Therefore, the above two terms can be combined into a single term like so:

$$2(((O - v1) \times E) \bullet (V \times E))t$$

Let us swap the positions of the two bracketed terms since it looks a bit nicer (our humble opinion). Thus, we have simplified the two terms that involve *t* to the following:

$$2((V \times E) \bullet ((O - v1) \times E))t$$

We have now isolated *t* on the right hand side of this expression and can see that the **b** coefficient that accompanies *t* in the middle term of the standard form must be:

$$b = 2((V \times E) \bullet ((O - v1) \times E))$$

Our equation now looks like this:

$$(V \times E)^2 t^2 +$$
$$((O - v1) \times E)^2 +$$
$$2((V \times E) \bullet ((O - v1) \times E))t$$

We are getting there! However, we know that in the standard form the t^2 term goes on the far left, the t term goes in the middle and the constant term **c** is written to the right. Therefore let us be consistent and write our equation that way too. Currently we have the t term (the middle term in the standard form) at the end of our equation rather than the middle, so let us swap the last two terms so that we have the term involving the squared unknown on the left and the non-squared unknown in the middle:

$$(V \times E)^2 t^2 + 2((V \times E) \bullet ((O - v1) \times E))t + ((O - v1) \times E)^2 = Radius^2$$

What is left must be the constant term **c** of the standard form, but we can see now that in its current form **c** must be:

$$c = ((O - v1) \times E)^2$$

Do not forget that in the standard form, we must have a zero on the RHS of the equals sign. That is no problem either. We can just subtract the squared radius from both sides of the equation leaving zero on the RHS and introducing **–Radius^2** on the LHS.

We now have our ray/cylinder intersection equation in the standard quadratic form:

$$(V \times E)^2 t^2 + 2(V \times E)((O - v1) \times E)t + ((O - v1) \times E)^2 - Radius^2 = 0$$

Finally, we have the **a**, **b**, and **c** coefficients of our quadratic in standard form and can simply plug them into our SolveCollision function to generate the t value of intersection (if one exists) between the ray and the cylinder. Below we show our final calculations for the coefficients in the more correct vector form. That is, that the terms inside the brackets are vectors, not single values, and as such a multiply must be replaced with the vector version of a multiply (a dot product operation). As you can see, any vectors that were in a squared form in the previous version of the equation have had there square sign removed by multiplying out the square. Therefore, **(VxE)^2** becomes **(VxE)•(VxE)** for example. This yields the final results of our coefficients exactly as we will calculate them in code.

$$a = (V \times E) \bullet (V \times E)$$
$$b = 2((V \times E) \bullet ((O - v1) \times E))$$
$$c = ((O - v1) \times E) \bullet ((O - v1) \times E) - (Radius * Radius)$$

You now know how to determine when the swept sphere collides with the edge of a triangle. If it does, the *t* value returned will describe (indirectly) the new position the center of the sphere should be moved to so that its surface is touching (but not intersecting) the edge.

You could say that for the test between the swept sphere and our triangle, the triangle was inflated by the radius of the sphere. The plane was shifted during the initial tests with the interior of the polygon, so we can liken this to inflating the depth of the polygon away from the plane so that it would have a height of Radius when viewed from any point on that plane. We have also learned that to test each edge, we should also inflate that edge by the same amount, forming a cylinder. It would seem that with these two techniques, we have fully thickened our polygon in all directions by a distance equal to the sphere's radius. This allows us safely reduce the swept sphere to a ray in our intersection tests. However, our inflating technique is still missing one thing which will cause gaps to be created. Treating the swept sphere as a ray without fixing these flaws would fail (causing the sphere to get stuck on the geometry). Let us now examine where these gaps exist that would cause collision testing to fail.

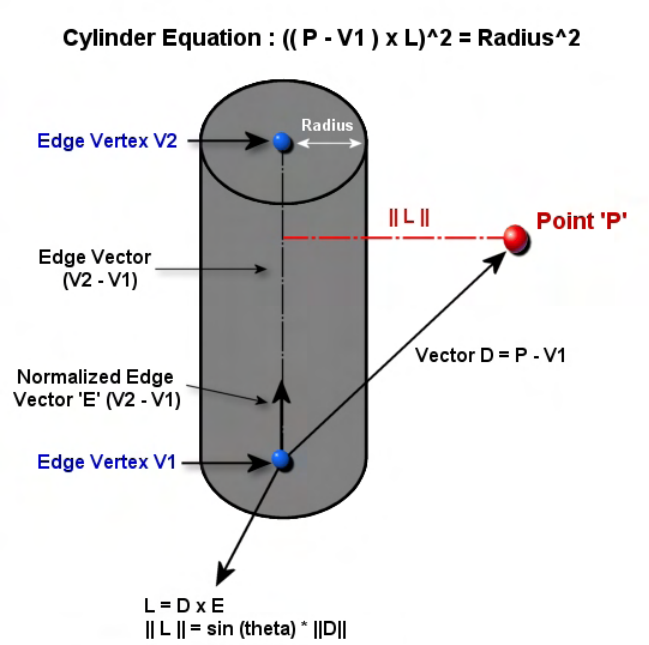


Figure 12.35

When we look at how we constructed a cylinder from our triangle edge, we must pay close attention to the type of cylinder we have created. The cylinder has no caps; it is just an infinite tube with no top or bottom. However, when testing our ray against this cylinder, we will project the intersection point on the surface of the cylinder onto the edge itself (the central spine of cylinder) to get that actual point on the infinite line where the intersection has occurred. If the intersection point projected onto vector **E** (the adjacent side) is outside the line segment (**V2-V1**) then it will be considered to have not intersected the edge. This means we have only bloated the edge in two dimensions and not three.

To understand the problem that we have, imagine that point **P** in Figure 12.35 was raised up so that it was only slightly higher than vertex **V2**. When the ray intersection point is projected onto the infinite line on which the edge is defined, it will be found to be higher than vertex **V2** and therefore, considered to be outside the line segment (**V2-V1**). In other words, the opposite side of the triangle (||L|| in Figure 12.35) is connecting with the central spine of the cylinder higher than vertex **V2**. The intersection with the infinite line is considered to occur beyond the edge extents when this is the case. However, we have not taken into consideration that the ends of each edge (the vertices themselves) should also be inflated by a distance equal to the sphere's radius to give complete padding around the edge end points. Where our edges end at the vertices, we should have domes filling these gaps. Figure 12.26 shows the gaps we currently have in our inflated triangle technique causing intersection tests to fail if not corrected. As you can see, we have not entirely padded the perimeter of the triangle as we should.

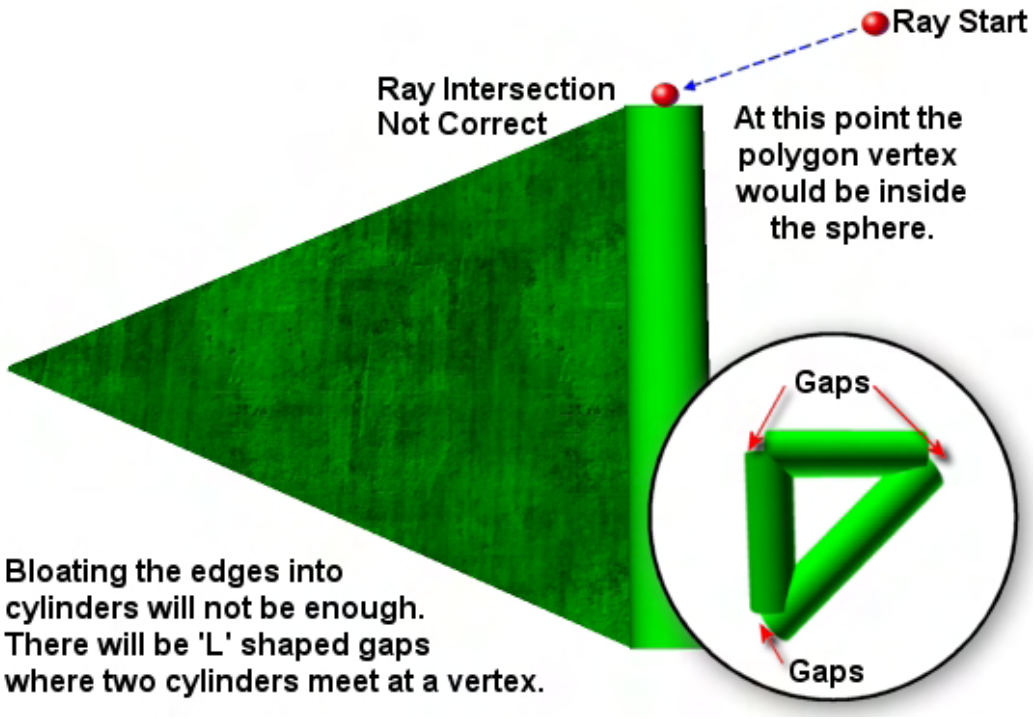


Figure 12.36

In Figure 12.36 we see a ray that is intersecting the top vertex of the edge currently being tested. Since we essentially discount any intersection with the surface of the cylinder that (in this diagram) is higher or lower than the two vertices that form the edge, the ray/cylinder test would return false for intersection in this case. However, it is clear that the ray collides right with the top of the cylinder, where the cap should be. Unfortunately the top vertex in the diagram has had no inflating applied and the intersection is undetected. Remembering that the *t* value returned from our intersection tests is used to calculate the new position of the **center** of the sphere, we can see that if we imagine the position at the end of the ray in Figure 12.36 to be allowed to be the new position of the sphere center, the sphere surface would be embedded in the polygon when its radius is taken into account.

If you look at the circular inset in Figure 12.36, you can see that were we to do nothing other than the edge tests so far described, there would be gaps at the ends of each edge. This can cause the swept sphere to become embedded around the areas of the vertices.

To fix this problem we have to perform an additional test with the vertices when the ray does not intersect the cylinder (i.e., the swept sphere did not intersect the interior of the edge). To remove the gaps at our vertex positions, we need to inflate those areas too so that we have a consistent radius thick buffer around the entire triangle perimeter.

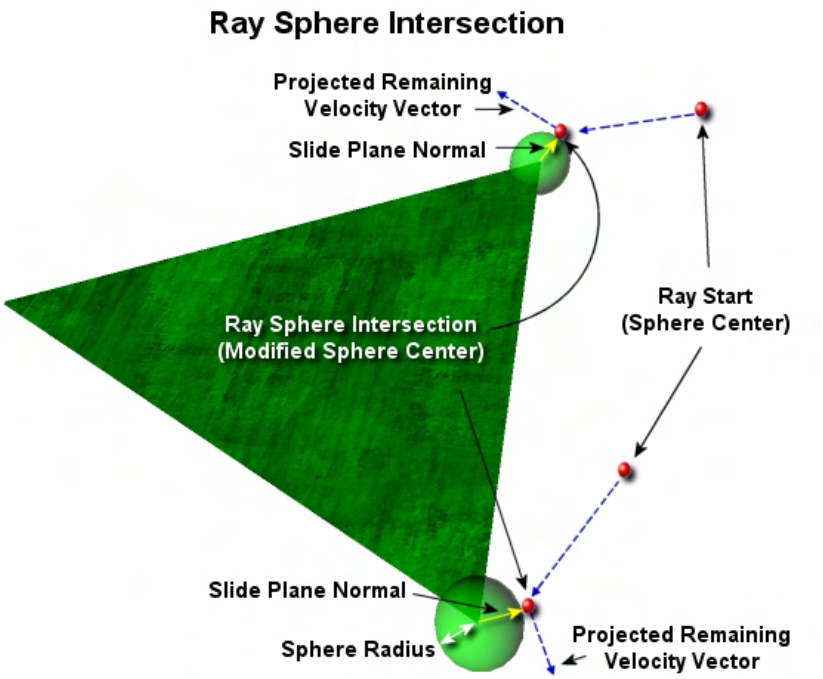


Figure 12.37

We can do this using a similar method to the previous test. We can just inflate each vertex by the radius of our sphere (in all directions). This turns each vertex in the edge into a sphere of the correct radius and allows us to treat our swept sphere as a simple ray. We can then calculate the intersection between the swept sphere and a vertex using a ray/sphere intersection test.

Thus, for each edge, we will first test the ray against the inflated edge (a cylinder) as previously discussed. If no intersection occurs, we will then test the same ray against each inflated vertex (a sphere) in that edge. This fully buffers the edge of the triangle currently being tested. In fact, if we examine the buffering effect with respect to both tests, we see that each edge is transformed into a capsule. In Figure 12.38, we see how the edge looks from the perspective of our ray when all intersection tests for that edge are taken into account. The first edge test inflates the edge into a cylinder and that is followed by a test on each vertex of the edge. As the vertices are inflated into spheres, we form a smooth capsule around the edge for our response system to slide around.

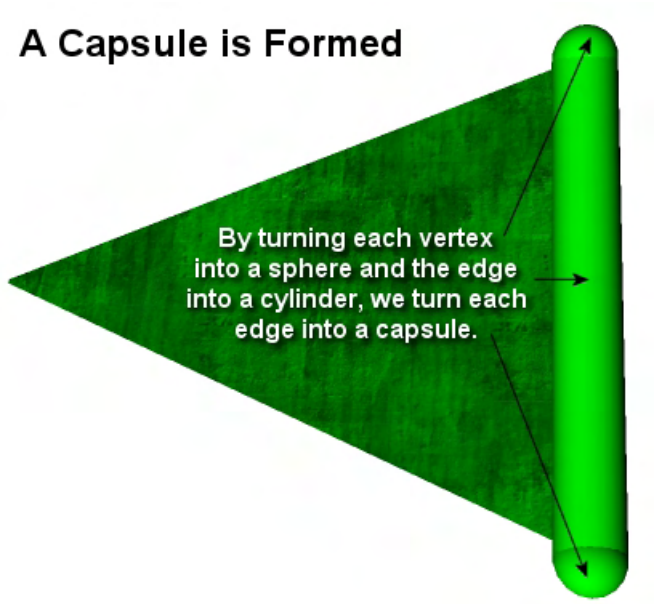


Figure 12.38

You will see in a moment that our edge intersection function will now have to solve (potentially) three quadratic equations. The first quadratic equation is for the intersection between a ray and a cylinder. Then we will have to perform two more intersection tests to determine if the ray (the swept sphere) intersects the spheres (the vertices) at each end of the capsule. As mentioned previously, the sphere tests only need to be performed if the ray has not collided with the cylinder.

When an intersection has occurred with one of the edge vertices (instead of the interior of the edge), the collision normal will be calculated by generating a normalized vector from the vertex (which is the center of the sphere) to the ray intersection point which is on the surface of the sphere (inflated vertex) as shown in Figure 12.39. This will generate a slide plane for the response phase allowing the ellipsoid to slide smoothly around the vertex after a collision.

Before we can look the code that determines these intersections, we still have a little more math to do. In order to test the ray against the sphere formed by each vertex, we must first learn how to substitute the ray equation into the sphere equation and manipulate the resulting quadratic equation into the standard form. Once we have learned how to do that, we will finally look at the complete set of code for the CCollision::SphereIntersectLineSegment function that performs all of these tests. This is the function that we called from CCollision::SphereIntersectTriangle for each edge in our triangle.

It should be noted that while the SphereIntersectLineSegment function is the main function call, for clarity the ray/sphere tests are performed in a separate helper function called SphereIntersectPoint. This function will be called from SphereIntersectLineSegment which directly handles only the ray/cylinder test.

12.7.2 Swept Sphere / Vertex Intersection

Figure 12.39 illustrates what we are trying to achieve with this intersection test. **Source** and **Dest** describe the initial and desired destination points for our swept sphere in this update. If we inflate the vertices into spheres equal to the radius of our swept sphere, we can treat our swept sphere as a ray. The ray will therefore intersect the sphere around the vertex giving us a new destination (**Modified Dest**) position for the center of the sphere that will be at an exact distance of Radius units from the actual vertex. This means, in reality, the surface of our sphere will be touching (but not intersecting) the vertex at this time. Therefore, the *t* value of intersection between the ray and the sphere surrounding the vertex

provides the furthest point along the velocity vector that the center of the sphere can move to before penetrating the vertex.

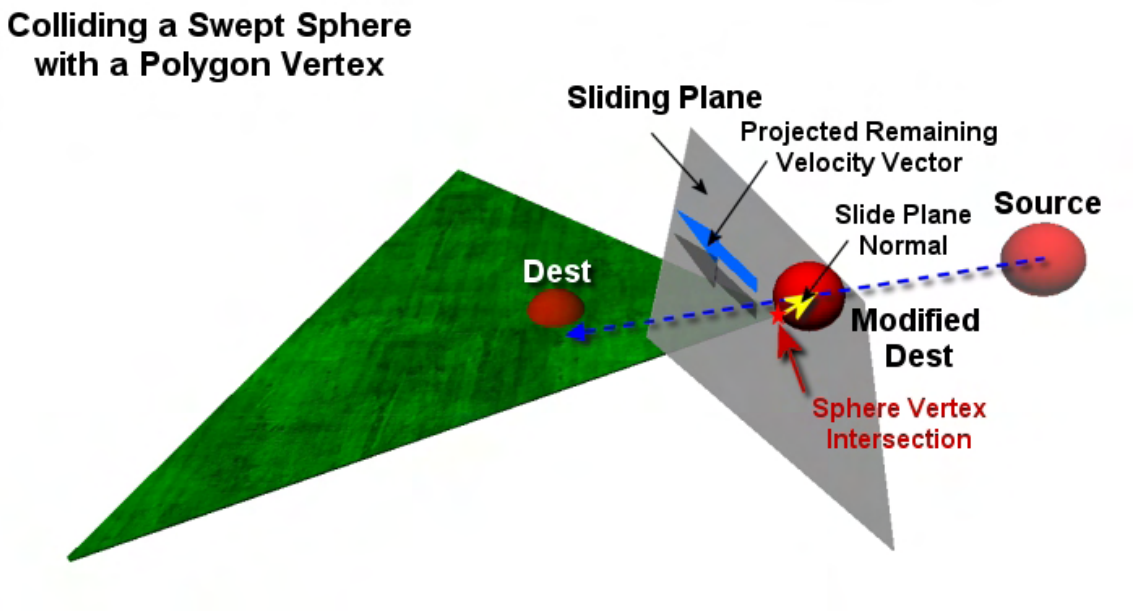


Figure 12.39

As can also be seen in Figure 12.39, the response step will require a sliding plane. The slide plane normal (collision normal) is calculated by returning a normalized vector pointing from the vertex position to the intersection point (**Modified Dest**) that serves as the new position of the center of the sphere.

Before coding can commence, we need to examine the ray/sphere intersection test. Therefore, we must start with the sphere equation, substitute the ray equation into that sphere equation, and then manipulate the resulting quadratic into the standard form so that we can call SolveCollision method to solve for *t*.

The equation for a sphere that we will use is shown below. **P** is any point that is on the surface of the sphere if the equation is true. **C** is the position of the center of the sphere and R is the radius of the sphere.

$$\|(P - C\|^2 = R^2$$

It is a rather simple equation to understand. If the squared length of a vector created from the position of the center of the sphere to an arbitrary point **P** is equal in length to the squared radius of the sphere, the point must be on the surface of the sphere. For any point not on the surface of the sphere, the equation will not be true.

We do not wish to simply test a point **P** using this equation; instead we want to find the position *t* along a ray at which the equation becomes true. If we can do that, we have found the point on the ray that is at a distance of Radius units from the sphere center and is therefore the exact time at which the ray pierces the sphere surface.

We need to substitute the equation of our ray:

$$O + tV$$

into the sphere equation in place of **P** and then solve for *t*. After substituting **O+tV** for **P** in the sphere equation we have the following equation:

$$\|O + tV - C\|^2 = R^2$$

We must first expand the LHS so it is no longer a squared expression. We know that the squared length of the vector **O+tV-C** can be obtained by multiplying this vector by itself, so let us multiply out the square on the LHS:

$$(O + tV - C) \bullet (O + tV - C) = R^2$$

Now we have to multiply two bracketed terms again, but this time each bracket contains three terms instead of two. Although the equations we used FOIL on previously had only two terms, it still works the same way. We just start with the first term in the left brackets working from left to right, and for each term on the left, we multiply it with each term on the right and finally sum the results of each test to get the final result.

Step 1: First we multiply the first term in the left brackets with the first term in the right brackets.

$$O \bullet O = O^2$$

Step 2: Now we multiply the first term in the left brackets with the second term on the right. In this step we rearranged the result to have *t* on the right. This is really a matter of preference thing at this stage. As *t* is the unknown, we will eventually be trying to isolate it as much as possible, and moving it to the right keeps the vectors (**O** and **V**) together in the term and keeps *t* separate. Remember *t* is just a scalar so we have not altered the evaluation of the expression by doing this; we simply rearranged the term more to our liking:

$$O \bullet tV = O \bullet Vt$$

Step 3: Multiply the first term in the left brackets with the last term in the right brackets. Since **O** · **–C** is the same as **–O** · **C**, we write it like this to make a tidier term with the sign on the left.

$$O \bullet -C = -O \bullet C$$

Let us have a look at what our equation looks like so far by summing the three results we have:

$$O^2 + O \bullet Vt - O \bullet C$$

Step 4: We have now performed all steps for the first term in the left brackets, so we do it all again now for the second term. That is, we will multiply each term in the right brackets with the second term in the left brackets (***t*V**).

$$tV \bullet O = O \bullet Vt$$

We rearrange the order of the result again to make it easier later when simplifying the equation. Since we already have an **O · Vt** term (see above), collecting like terms will be very straightforward.

Step 5: Now we multiply the second term in the left brackets with the second term in the right brackets.

$$tV \bullet tV = tV^2 = t^2V^2 = V^2t^2$$

It is pretty clear now that this is definitely a quadratic-- it has a term that involves the unknown *t* raised to the second power.

Step 6: Next we multiply the second term in the left brackets with the last term in the right brackets, yielding the following:

$$tV \bullet -C = -C \bullet Vt$$

At this point we have multiplied the first and second terms in the left brackets with all terms in the right brackets. We could say at this point that we have two thirds of our equation's LHS complete. After the six steps we have completed, we have six terms which, when added together, yields this current LHS:

$$O^2 + O \bullet Vt - O \bullet C + O \bullet Vt + V^2t^2 - C \bullet Vt$$

Step 7: Finally, we have to multiply the final term in the left brackets with each term in the right brackets, giving us the final three terms for our expanded equation. First, we multiply the last term in the left brackets with the first term in the right brackets. We rearrange this to **–O · C** since we already have a term in this shape:

$$-C \bullet O = -O \bullet C$$

S**tep 8:** Next we multiply the last term on the left with the middle term on the right.

$$-C \bullet tV = -C \bullet Vt$$

Step 9: And lastly, we multiply the two final terms of each bracket. This is the multiplication of two negatives **–C** and **–C** giving the positive result $\mathbf{C^2}$.

$$-C \bullet -C = C^2$$

Let us now see the final (and quite large) left hand side of our fully expanded quadratic equation:

$$O^2 + O \bullet Vt - O \bullet C + O \bullet Vt + V^2t^2 - C \bullet Vt - O \bullet C - C \bullet Vt + C^2$$

That initially looks pretty daunting, but there are lots of like terms we can collect. For example, look at the third term **(-O ·C**) and the and seventh term (also **–O · C**). As these are identical and we are just subtracting **–O · C** from the LHS twice, the equation can be collected in the third term like so:

$$O^2 + O \bullet Vt - 2(O \bullet C) + O \bullet Vt + V^2t^2 - C \bullet Vt - C \bullet Vt + C^2$$

We also have many terms containing *t* (not t^2) and we should collect these to form the middle term of our quadratic in standard form. If we collect *t* terms and then isolate *t*, we will also have our **b** coefficient for our standard form quadratic.

Let us just collect all the terms with *t* in them and line them up together:

$$O \bullet Vt + O \bullet Vt - C \bullet Vt - C \bullet Vt$$

We can see that we are adding together two (**O · Vt**) terms on the LHS and also subtracting the term (**C · Vt**) twice from the LHS. Thus, we can collapse these four terms into two terms:

$$2(O \bullet Vt) - 2(C \bullet Vt)$$

That looks a lot nicer, but we still need this arranged into a single term with an isolated *t*, so that we have our **b** coefficient available. Finding a common factor will solve all of our problems. With common factors, if there is some shared multiple across all terms, that multiple can be moved outside the brackets. In our case, we see that both terms have the number **2** and both terms have ***t***. Thus, both **2** and ***t*** are common factors which can be placed outside a set of brackets that contains the rest of the expression (we call this "factoring out" the 2 and the t):

$$2t(O \bullet V - C \bullet V)$$

Using the same logic, we can see that the two terms inside the brackets both contain **V**. Therefore, we can move **V** outside a new set of brackets that contain **O - C**.

$$2t(V \bullet (O - C))$$

Notice how we are careful to preserve the dot product sign since both **O - C** and **V** are vectors and should be vector multiplied.

Finally, let us put *t* over on the right hand side of the outer brackets to isolate it more clearly. This does not change the evaluation of the expression, but it does provide clarity for showing us what the **b** coefficient is going to be in standard form (**2 * (V · (O – C))**:

$$2(V \bullet (O - C))t$$

Plugging this back into our original equation in place of all the *t* terms we had before gives us the following LHS:

$$O^2 + 2(V \bullet (O - C))t - 2(O \bullet C) + V^2 t^2 + C^2$$

Looking at this equation we can instantly see what our **a** coefficient is going to be. It is the value accompanying $\boldsymbol{t^2}$. We can see that this is $\mathbf{V^2}$. Let us move that term to the left of the equation so that it assumes its rightful position in the standard form:

$$V^2 t^2 + O^2 + 2(V \bullet (O - C))t - 2(O \bullet C) + C^2$$

We can also see what the **b** coefficient in standard form is going to be. It is the expression accompanying ***t*** in the non-squared term: $\mathbf{2(V \cdot (O - C))}$. Therefore, let us move the *t* term to its rightful position in the standard form:

$$V^2 t^2 + 2(V \bullet (O - C))t + O^2 - 2(O \bullet C) + C^2$$

Thus, we can see that our **c** coefficient (so far) must be $\mathbf{O^2 - 2(O \cdot C) + C^2}$. That expression is a little lengthy. But wait just a moment. Do we recognize any pattern in the term? Indeed we do. It matches the identity:

$$O^2 - 2OE + E^2 = (O - E)^2$$

In our current constant term we can see that **O** = **O** and **E** = **C** and thus, the identity tells us that it can it also be written in its more compact form $\mathbf{(O - C)^2}$. Below we show the full equation (RHS as well) so far:

$$V^2 t^2 + 2(V \bullet (O - C))t + (O - C)^2 = R^2$$

We are almost there! The standard form quadratic must have zero on the RHS, so we must subtract $\mathbf{R^2}$ from both sides of the equation. This gives us the final ray/sphere intersection quadratic equation in standard form:

$$V^2 t^2 + 2(V \bullet (O - C))t + (O - C)^2 - R^2 = 0$$

We can write this slightly differently to reflect a pattern that is more convenient for programmers (it is the same equation though):

$$(\mathrm{V} \bullet \mathrm{V})\mathrm{t}^2 + 2(\mathrm{V} \bullet (\mathrm{O} - \mathrm{C}))\mathrm{t} + (\mathrm{O} - \mathrm{C}) \bullet (\mathrm{O} - \mathrm{C}) - \mathrm{R} * \mathrm{R} = 0$$

Therefore, the final a, b, and c coefficients that we were searching for (which will be fed into our SolveCollision function to solve for *t*) are:

$$a = V \bullet V$$

$$b = 2 * (V \bullet (O - C))$$

$$c = (O - C) \bullet (O - C) - R * R$$

12.8 The SphereIntersectLineSegment Function

We are now ready to put everything we have learned in the last two sections into practice and write a function that will determine if a swept sphere intersects a triangle edge. Remember, this function is called three times from the SphereIntersectTriangle function (once per edge). Since the function is passed the current closest *t* value that has been found so far (tMax), it will only return true if an intersection happens with the edge at a distance closer than one already found. If this happens, the current *t* value will be overwritten with the new *t* value.

This first version of the function shows the code for the equations we have discussed. You will see in a moment that we will add some additional code to this function to correct the situation where the sphere may already be embedded in the edge prior to the intersection test being performed. Since this will complicate things slightly, we will first show the pure code and then worry about the special case handling.

```
bool CCollision::SphereIntersectLineSegment( const D3DXVECTOR3& Center,
                                             float Radius,
                                             const D3DXVECTOR3& Velocity,
                                             const D3DXVECTOR3& v1,
                                             const D3DXVECTOR3& v2,
                                             float& tMax,
                                             D3DXVECTOR3& CollisionNormal )
{
    D3DXVECTOR3 E, L, X, Y, PointOnEdge, CollisionCenter;
    float       a, b, c, d, e, t, n;

    // Setup the equation values
    E = v2 - v1;              // E = triangle Edge V2-V1
    L = Center - v1;          // L = Hypotenuse ( Vector from V1 to Ray Origin )
    e = D3DXVec3Length( &E );  // e = length of triangle edge

    if ( e < 1e-5f ) return false;  // Safety test:
                                    // If triangle edge has no length return false

    E /= e;             // Normalize E ( Unit length Adjacent )

    // Generate cross values
```

```
    D3DXVec3Cross( &X, &L, &E );      // ||X|| = Length of Opposite side
                                      // ( Hypotenuse * sin a )
                                      //    = Distance from Ray Origin to Edge
    D3DXVec3Cross( &Y, &Velocity, &E );  //  Y = Velocity x E

    // Setup the input values for the quadratic equation
    a = D3DXVec3LengthSq( &Y );           // ( VxE ) dot ( VxE )
    b = 2.0f * D3DXVec3Dot( &X, &Y );     // 2 *(RayOrigin - v1 x E) dot ( V x E )
    c = D3DXVec3LengthSq( &X ) - (Radius*Radius); // ( RayOrigin -v1 x E )^2 – R*R

    // Solve the quadratic for t
    if ( !SolveCollision(a, b, c, t) ) return false;
```

If you cross reference the way we calculate the a, b and c coefficients above, you should see that this matches exactly what we determined they should be when we arranged the ray/cylinder equation into a standard form quadratic equation. We simply plug these values into the SolveCollision function which solves the quadratic equation using the quadratic formula. Remember that the SolveCollision function will return in its output parameter *t*, the smallest positive solution to the quadratic. Therefore, if the function returns true, the ray did intersect the cylinder and local variable *t* will contain the first point of intersection in front of the ray origin. If no intersection is found, we can return false from this function immediately and pass program flow back to the calling function SphereIntersectTriangle (which will then proceed to test the other edges for intersection).

Notice that we have added a test that checks the length of the edge vector (*e*). If *e* is zero (with tolerance) then we have been passed a degenerate edge and return immediately.

The last half of this function is shown below. It first tests to see if the *t* value returned from SolveCollision is larger than the value passed in the tMax parameter. At this point, tMax will contain the closest *t* value found while testing all previous triangles/edges. We are only interested in finding the closest intersecting triangle, so if we have previously detected a collision with a triangle that happens closer to the ray origin than is described by the current *t* value we have just had returned, we can return immediately since we are not interested in this intersection. In that case, we have already hit something that is in front of it.

```
     if ( t > tMax ) return false; // Intersection distance is larger
                                   // than one already found so return false
```

If no closer intersection has yet been found, then we need to calculate the intersection point along the ray. This will also be the point of intersection on the surface of the cylinder. After testing all remaining triangles, if our detection phase determines that this is the closest intersection, this position (stored in the local CollisionCenter variable) will describe the new center position of our sphere (i.e., ellipsoid) such that the surface is just contacting the edge. Calculating this position along the ray is easy now that we have our *t* value.

```
    // Use returned t value to calculate intersection position along velocity ray
    CollisionCenter = Center + Velocity * t;
```

Now that we have a temporary position on the cylinder surface where intersection will occur, we create a vector from the base of the cylinder (v1) to this position. This forms the hypotenuse of a triangle with the edge as its adjacent leg. If we dot this vector with unit length edge vector E, we will scale the length of the hypotenuse by the cosine of the angle between them. This gives us the length of the adjacent leg. This length will describe the exact distance along the edge from vertex v1 to the point where the intersection with the edge has occurred. This allows us to test whether this point is between vertices v1 and v2. All we are essentially doing is projecting the collision point on the cylinder surface onto the infinite line on which the edge is contained.

```
    // Project this onto the line containing the edge ( v2-v1 )
    d = D3DXVec3Dot( &(CollisionCenter - v1), &E );

    if ( d < 0.0f ) // The intersection with the line happens before
                    // the first vertex v1, so test vertex v1 too
        return SphereIntersectPoint( Center, Radius, Velocity,
                                     v1, tMax, CollisionNormal);
    else
    if ( d > e )     // The intersection with line happens past
                     // the last vertex v2 so test vertex v2 too
        return SphereIntersectPoint( Center, Radius, Velocity,
                                     v2, tMax, CollisionNormal);
```

Once we have *d* (the distance from v1 to the point of intersection along the infinite line that contains the edge), we test to see if it is smaller than zero. If it is, then the intersection with the edge has occurred before the first vertex and therefore, the ray is considered non-intersecting with our main cylinder. In this case, we call the SphereIntersectPoint point function to test for intersection against vertex v1. However, if the distance to the point of edge intersection (*d*) from vertex v1 is larger than the length of the edge (*e*), an intersection may have occurred with the second vertex (v2), so the same intersection test is done for vertex v2. Notice that either way we return at this point.

The SphereIntersectPoint function is passed all the information it needs to perform its tasks properly. It is passed the tMax value so that it will store a new *t* value and return true only if an intersection occurs that is closer than the current value in tMax. It is also passed the output parameter CollisionNormal so that it can calculate a collision normal and return it back to the caller.

Finally, if we get past this part, we know we have an intersection with the edge cylinder closer than any that has previously been found. All our collision detection phase wants us to return at this point (and the same was true for the calling function SphereIntersectTriangle) are two pieces of information: the *t* value tMax and a collision normal (slide plane normal). We will calculate the normal by building a normalized vector from the point of intersection on the actual edge, to the point at which the ray intersected the surface of the cylinder (as described earlier). What we are actually doing then, is generating a vector from the point of intersection on the edge, to the center of the swept sphere at the time of intersection.

We calculate the point of intersection on the edge by scaling the unit length edge vector E by the distance to the point of intersection from the start of the edge (*d*) and adding the resulting vector to the edge start vector (v1).

```
    // Calculate the actual Point of contact on the line segment
    PointOnEdge = v1 + E * d;

    // We can now generate our normal, store the interval and return
    D3DXVec3Normalize( &CollisionNormal, &(CollisionCenter - PointOnEdge) );
    tMax = t;

    // Intersecting!
    return true;

}
```

With the collision normal generated, we also overwrite the value currently stored in tMax with the new closer *t* value we have found and then return true. We will look at the complete code to this function in one block in a moment after we discuss something else that must be added.

12.8.1 Ray / Cylinder Intersection: Embedded Object Correction

Recall that the first intersection test we call in SphereIntersectTriangle is the SphereIntersectPlane function. This is a simple ray/plane test where the swept sphere is reduced to a ray by shifting the plane. However, recall that we performed tests within that function to determine if the sphere was already embedded in the plane prior to the test being performed. When this is the case, we have very few options other than to calculate a new position for the sphere so that it is no longer intersecting. To address this issue we calculated the distance from the ray origin to the plane (that was behind it) and then moved the ray origin back along the normal so that it was sitting on the plane instead of behind it. This worked well because the ray origin represents the center of the sphere, and when the sphere center is located on the shifted plane, its surface is touching the actual plane (and is no longer embedded).

Note that when this embedded case correction had been performed, the *t* value returned was not a positive parametric ray position in the range [0.0, 1.0]. It was a negative value describing the actual world space distance to the plane. In other words, the *t* value describes to the detection phase how far the sphere should be moved backwards so that it is no longer embedded.

It seems reasonable that we should perform these same embedded tests and corrections when performing our cylinder and sphere tests. For example, it is quite possible that our swept sphere's surface may already be embedded in the triangle edge prior to the intersection test being performed.

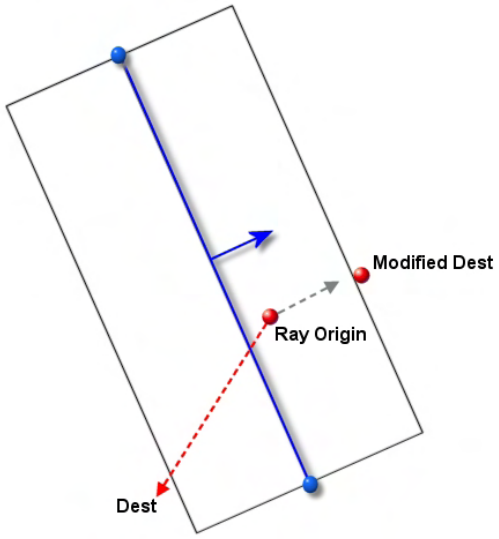


Figure 12.40

In Figure 12.40 we can see that the **Ray Origin** is inside the cylinder formed from the edge prior to the test being performed. **Dest** illustrates the position that the sphere is being requested to move to. In this case (like the plane case), we just ignore the velocity vector

and move the sphere center (ray origin) in the direction of the edge normal so that it is no longer inside the cylinder and is instead located on the surface. Although we do not have the normal of the edge at this point, we can easily calculate it by projecting the ray origin onto the edge. We can then create a vector from the projected point on the edge to the original ray origin and make it unit length. Once we have the unit length edge normal, we can shift the ray origin along it. The distance we need to shift is equal to the radius of the cylinder minus the distance from the ray origin to the edge.

When this is the case, our SphereIntersectLineSegment function will not even have to bother solving the quadratic equation. We simply calculate the negative distance from the ray origin to the surface of the cylinder and return this as the *t* value to the detection phase. The collision normal returned will be the normal of the edge.

> **Note:** It is important to remember that none of our intersection techniques return or update the actual sphere position. That is done by the parent detection function (which we have not yet covered) that is called by our CollideEllipsoid function. All the intersection functions return is a slide plane normal and a *t* value of intersection. If negative, the *t* value will describe the actual world space distance the sphere must be moved so it is no longer embedded.

The question has to be asked, how do detect whether our ray origin is already within the cylinder? As it happens, we already have this answer stored in the **c** coefficient of our quadratic equation. Recall that the **c** coefficient was calculated like so:

$$c = ((O - v1) \times E) \bullet ((O - v1) \times E) - (Radius * Radius)$$

Where:

O = Sphere Center (Ray Origin)
v1 = First vertex in edge
E = Unit length edge vector

Therefore,

((O-v1)xE)•((O-v1)xE) = Squared Distance from ray origin to edge

Since we subtract the squared radius when calculating **c**, this value will be zero if the ray origin is on the surface of the cylinder, positive if the ray origin is outside the cylinder, and negative if the ray origin is inside the cylinder (i.e., distance between the ray origin and the edge is smaller than the radius).

So after we have calculated the coefficients for the ray/cylinder quadratic equation, but before we have solved it, we can test **c**. If it is negative, we can execute the following code in lieu of solving the quadratic equation.

```
    if ( c < 0.0f )
    {
        // Product Ray Origin onto Edge 'E'
        d = D3DXVec3Dot( &L, &E );

        // Is this before or after line start?
```

```
        if (d < 0.0f)
        {
            // The point is before the beginning of the line,
            // test against the first vertex instead
            return SphereIntersectPoint( Center, Radius, Velocity,
                                         v1, tMax, CollisionNormal );

        } // End if before line start
        else if ( d > e )
        {
            // The point is after the end of the line,
            // test against the second vertex
            return SphereIntersectPoint( Center, Radius, Velocity,
                                         v2, tMax, CollisionNormal );

        } // End if after line end
```

The first section of this code calculates *d*. This is calculated by projecting the length of the vector from v1 to the ray origin (L) onto the unit length edge vector E. This gets us the distance from vertex v1 to the ray origin (hypotenuse) projected onto vector E (adjacent). The length of this adjacent side L describes the distance from vertex v1 to the ray origin projected onto the line containing the edge. If this distance value is smaller than zero or larger than the length of the edge (*e*), then the sphere is not embedded in the actual cylinder, but may be embedded in the vertices (the spheres). Therefore, if *d* is negative, we perform an embedded correction test for v1, or if $d > e$, we perform the embedded correction test against v2. In both cases, we will let the SphereIntersectPoint function sort out the embedded correction.

If *d* contains a distance value between v1 and v2 then the ray origin is inside the cylinder and we need to correct it. As shown below, we first calculate the point on the actual edge. This is the ray origin projected onto the line that contains edge (v2-v1). Since we already have *d* (the distance to this projected point along the edge from v1), we can calculate the actual point on the edge by adding the vector E**d* to v1.

```
        else
        {
            // Point within the line segment
            PointOnEdge = v1 + E * d;

            // Generate collision normal
            CollisionNormal = Center - PointOnEdge;
            n = D3DXVec3Length( &CollisionNormal );
            CollisionNormal /= n;

            // Calculate negative t value as actual distance
            // by subtracting distance from edge to ray origin
            // ( which is smaller than radius ) the radius.
            t = n - Radius;

            if (tMax < t) return false;

            // Store t and return
            tMax = t;

            // Edge Overlap
```

```
            return true;

        } // End if inside line segment

    } // End if sphere inside cylinder
```

After we have the point on the edge, we now have to calculate the edge normal. We do this by creating a vector from the point on the edge to the ray origin. We then record the length of this vector in *n*, which describes the distance from the ray origin to the actual edge. Next we divide the collision normal by *n* to normalize it and we have the vector we will use as our slide plane normal later on. Now we just have to calculate the *t* value (shown above). Since *n* describes the distance from the ray origin to the edge, and this is now guaranteed to be smaller than the radius of the cylinder, subtracting the radius from *n* will give us a negative *t* value describing the distance from the ray origin to the surface of the cylinder. Because this *t* value is negative, it is guaranteed to overwrite any non-embedded *t* value that has been recorded in tMax while checking previous triangles. We record this value in tMax and return. Our function will have returned the *t* value and collision normal back to the parent detection function.

Let us now merge all of these concepts together to create the final code to our CCollision::SphereIntersectLineSegment function (complete with embedded case correction).

12.8.2 The Final SphereIntersectLineSegment Function

```
bool CCollision::SphereIntersectLineSegment( const D3DXVECTOR3& Center,
                                             float Radius,
                                             const D3DXVECTOR3& Velocity,
                                             const D3DXVECTOR3& v1,
                                             const D3DXVECTOR3& v2,
                                             float& tMax,
                                             D3DXVECTOR3& CollisionNormal )
{
    D3DXVECTOR3 E, L, X, Y, PointOnEdge, CollisionCenter;
    float       a, b, c, d, e, t, n;

    // Setup the equation values
    E = v2 - v1;              // E = triangle Edge V2-V1
    L = Center - v1;          // L = Hypotenuse ( Vector from V1 to Ray Origin )
    e = D3DXVec3Length( &E ); // e = length of triangle edge

    if ( e < 1e-5f ) return false;  // Safety test…
                                    // if triangle edge has no length return false

    E /= e;             // Normalize E ( Unit length Adjacent )

    // Generate cross values
    D3DXVec3Cross( &X, &L, &E );      // ||X|| = Length of Opposite side
                                      // ( Hypotenuse * sin a )
                                      //    = Distance from Ray Origin to Edge
    D3DXVec3Cross( &Y, &Velocity, &E );   //  Y     = Velocity x E

    // Setup the input values for the quadratic equation
```

```
    a = D3DXVec3LengthSq( &Y );          // ( V x E ) dot ( V x E )
    b = 2.0f * D3DXVec3Dot( &X, &Y );    // 2*( RayOrigin-v1 x E ) dot ( V x E )
    c = D3DXVec3LengthSq( &X ) - (Radius*Radius); // ( RayOrigin-v1 x E )^2 – R*R

    if ( c < 0.0f )
    {
        // Product Ray Origin onto Edge 'E'
        d = D3DXVec3Dot( &L, &E );

        // Is this before or after line start?
        if (d < 0.0f)
        {
            // The point is before the beginning of the line,
            // test against the first vertex instead
            return SphereIntersectPoint( Center, Radius, Velocity, v1,
                                         tMax, CollisionNormal );

        } // End if before line start
        else if ( d > e )
        {
            // The point is after the end of the line,
            // test against the second vertex
            return SphereIntersectPoint( Center, Radius, Velocity,
                                         v2, tMax, CollisionNormal );

        } // End if after line end
        else
        {
            // Point within the line segment
            PointOnEdge = v1 + E * d;

            // Generate collision normal
            CollisionNormal = Center - PointOnEdge;
            n = D3DXVec3Length( &CollisionNormal );
            CollisionNormal /= n;

            // Calculate negative t value as actual distance
            // by subtracting distance from edge to ray origin
            // ( which is smaller than radius ) the radius.
            t = n - Radius;

            if (tMax < t) return false;

            // Store t and return
            tMax = t;

            // Edge Overlap
            return true;

        } // End if inside line segment

    } // End if sphere inside cylinder

    // If we are already checking for overlaps, return
    if ( tMax < 0.0f ) return false;

    // Solve the quadratic for t
```

```
    if ( !SolveCollision(a, b, c, t) ) return false;

    if ( t > tMax ) return false; // Intersection distance is larger
                                  // than one already found so return false

    // Use returned t value to calculate intersection position along velocity ray
    CollisionCenter = Center + Velocity * t;

    // Project this onto the line containing the edge ( v2-v1 )
    d = D3DXVec3Dot( &(CollisionCenter - v1), &E );

    if ( d < 0.0f ) // The intersection with the line happens
                    // before the first vertex v1 so test vertex v1 too
        return SphereIntersectPoint( Center, Radius, Velocity,
                                     v1, tMax, CollisionNormal);
    else
    if ( d > e )     // The intersection with line happens
                     // past the last vertex v2 so test vertex v2 too
        return SphereIntersectPoint( Center, Radius, Velocity,
                                     v2, tMax, CollisionNormal);

    // Calculate the actual Point of contact on the line segment
    PointOnEdge = v1 + E * d;

    // We can now generate our normal, store the interval and return
    D3DXVec3Normalize( &CollisionNormal, &(CollisionCenter - PointOnEdge) );
    tMax = t;

    // Intersecting!
    return true;

}
```

One line worth drawing your attention to is highlighted in bold:

```
  // If we are already checking for overlaps, return
  if ( tMax < 0.0f ) return false;
```

If we have gotten to this point in the function, the swept sphere is not embedded in the edge currently being tested. Normally, a quadratic equation would then be solved to find the *t* value of intersection along the ray. However, if the tMax value passed in is already negative, then we will not waste time determining the intersection point along the ray since we have already found a triangle (in a previous test) in which the swept sphere is embedded. From that point on, we are only interested in finding triangles for which the swept sphere is potentially *more* embedded.

You can think about our intersection routines as working in one of two modes. Until we find an embedded triangle, we solve quadratic equations to find a closest *t* value. In this mode, the *t* value will be a parametric distance along the ray in the range of [0.0, 1.0]. However, as soon as we find an embedded triangle, tMax is set to a negative distance value which describes how to correct the situation by moving the sphere. At this point, the intersection tests are switched into what we might call ‘Embedded Correction Mode’. For the remainder of our tests, we are only interested in other triangles that the sphere is embedded in. Therefore, tMax becoming negative switches into this Embedded

Correction Mode, and this can be done by the SphereIntersectPlane, SphereIntersectLineSegment and SphereIntersectPoint functions.

We have just one more function to write before we have completed the methods needed to sweep a sphere against a triangle. The SphereIntersectLineSegment function calls a function called SphereIntersectPoint. This is the final missing piece we need to discuss. We will then have covered every separate intersection test called directly or indirectly by the SphereIntersectTriangle function.

12.9 The SphereIntersectPoint Function

SphereIntersectPoint is called by the previous function if no intersection is found between the ray and the cylinder formed by the edge. This additional test looks to see if the ray intersects the sphere that we construct around each vertex (which we do to fully pad the triangle edge). The code just calculates the **a**, **b**, and **c** coefficients of the quadratic equation and passes them into our SolveCollision. It will return either true or false depending on whether the ray intersects the sphere. If the *t* value returned is greater than the one we currently have in the tMax parameter, we ignore it and return false for intersection; otherwise, we overwrite the value stored in tMax with our new, closer intersection *t* value so that it can be returned to the caller. We then calculate this new position along the ray (which is on the surface of the inflated sphere at a distance of Radius units from the vertex). Finally, we calculate the collision normal by generating a unit length vector from the vertex to the intersection point on the surface of the sphere.

As with the previous function, if that was all we had to worry about then the function would be extremely small. However, we also have to cater for the case where the ray origin is already inside the sphere surrounding the vertex. If it is then we will not bother solving the quadratic equation and will simply calculate the distance we need to move the sphere center position (the ray origin) along the collision normal so that it is no longer penetrating the vertex. In other words, just as we have done in all of our intersection functions, we will return a negative *t* value which describes the actual distance (not parametric) we have to move the sphere center position back so that it is correctly positioned at a distance of Radius from the vertex, just contacting the inflated sphere surrounding that vertex.

We will look at this function in a few sections. In the first part of the function we calculate the **a**, **b**, and **c** coefficients of the quadratic equation using the mathematics we discussed earlier.

```
bool CCollision::SphereIntersectPoint( const D3DXVECTOR3& Center,
                                       float Radius,
                                       const D3DXVECTOR3& Velocity,
                                       const D3DXVECTOR3& Point,
                                       float& tMax,
                                       D3DXVECTOR3& CollisionNormal )
{
    D3DXVECTOR3 L, CollisionCenter;
    float       a, b, c, l, l2, t;

    // Setup the equation values
    L  = Center - Point;
```

```
    l2 = D3DXVec3LengthSq( &L );

    // Setup the input values for the quadratic equation
    a = D3DXVec3LengthSq( &Velocity );
    b = 2.0f * D3DXVec3Dot( &Velocity, &L );
    c = l2 - (Radius * Radius);
```

We only need to solve this equation if the **c** coefficient is non-negative. The **c** coefficient contains the squared length of a vector from the ray origin to the vertex, minus the squared radius of the sphere. If it is negative, then the distance from the vertex to the ray origin is smaller than the radius of the inflated sphere surrounding the vertex. This tells us that the ray origin is already inside the sphere. In reality, this means the vertex is embedded inside our swept sphere and we need to back the swept sphere away from the vertex so that this is no longer the case. The code that calculates the *t* value to accomplish this is shown below.

```
    // If c < 0 then we are overlapping, return the overlap
    if ( c < 0.0f )
    {
        // Remember, when we're overlapping we have no choice
        // but to return a physical distance (the penetration depth)
        l = sqrtf( l2 );
        t = l - Radius;

        // Outside our range?
        if (tMax < t) return false;

        // Generate the collision normal
        CollisionNormal = L / l;

        // Store t and return
        tMax = t;

        // Vertex Overlap
        return true;

    } // End if overlapping
```

We store the distance from the ray origin (the center of the swept sphere) to the vertex (the center of our inflated sphere) in local variable *l*. Since we know that this value is already smaller than the radius of the inflated sphere (otherwise we would not be in this code block) we can subtract the radius from this value to get the negative distance (*t*) we need to move the ray origin so that it is located on the surface of the inflated sphere (i.e., the vertex is now just touching the surface of the swept sphere). If this *t* value is larger than the one already stored in tMax, it means that we have found another intersection in which the swept sphere is more deeply embedded. So we can ignore the current one and return false for intersection. Otherwise, we calculate the collision normal as the unit length vector from the vertex to the ray origin. We then store our *t* value in tMax and return true for intersection. From this point on, our detection phase will only be intersected in finding embedded polygons that are more deeply embedded than this one.

Beneath this code block is the code that gets executed when the current vertex is not already embedded in our swept sphere. Before solving the quadratic, we first test whether the current value stored in tMax

is negative. If it is, it means a previous intersection test found an embedded situation and we are no longer interested in anything other than finding another triangle, edge, or vertex that is more deeply embedded than the current one. So we return false for intersection.

If that is not the case, we are still searching for a closer intersection along the ray and we call our SolveCollision function to see whether the ray intersects the sphere surrounding the vertex. If not, we return false; otherwise, we test the returned *t* value and return false if it is found to be greater than the value we already have stored in tMax. We do this because, although we have found an intersection, it is happening at a further distance along the ray from the origin than an intersection we found on a previous test. Remember, our collision detection phase is only interested in finding the closest colliding triangle.

```
    // If we are already checking for overlaps, return
    if ( tMax < 0.0f ) return false;

    // Solve the quadratic for t
    if (!SolveCollision(a, b, c, t)) return false;

    // Is the vertex too far away?
    if ( t > tMax ) return false;
```

Finally, we calculate the point of intersection along the ray and use this to create a collision normal. The normal is generated by normalizing a vector from the vertex to the intersection point on the surface of the inflated sphere surrounding the vertex.

```
    // Calculate the new sphere position at the time of contact
    CollisionCenter = Center + Velocity * t;

    // We can now generate our normal, store the interval and return
    D3DXVec3Normalize( &CollisionNormal, &(CollisionCenter - Point) );

    tMax = t;

    // Intersecting!
    return true;
}
```

As can be seen above, we store the new *t* value in tMax so the calling function can pass it back to the collision detection phase and use it for comparisons in future triangle intersection tests.

And there we have it. We have discussed all the intersection techniques that we need to understand in order to implement this collision system. Primarily we have described exactly how we test a swept sphere against a triangle. First, we test against the plane of the triangle, then we test against the interior of the triangle, and finally tests are performed against each edges and vertices of the triangle.

Before leaving this section, here is another look at the SphereIntersectTriangle function:

```
bool CCollision::SphereIntersectTriangle(  const D3DXVECTOR3& Center,
                                           float Radius,
                                           const D3DXVECTOR3& Velocity,
```

```
                                          const D3DXVECTOR3& v1,
                                          const D3DXVECTOR3& v2,
                                          const D3DXVECTOR3& v3,
                                          const D3DXVECTOR3& TriNormal,
                                          float& tMax,
                                          D3DXVECTOR3& CollisionNormal )
{

    float       t = tMax;
    D3DXVECTOR3 CollisionCenter;
    bool        bCollided = false;

    // Find the time of collision with the triangle's plane.
    if ( !SphereIntersectPlane( Center, Radius, Velocity, TriNormal, v1, t ))
        return false;

    // Calculate the sphere's center at the point of collision with the plane
    if ( t < 0 )
        CollisionCenter = Center + (TriNormal * -t);
    else
        CollisionCenter = Center + (Velocity * t);

    // If this point is within the bounds of the triangle,
    // we have found the collision
    if ( PointInTriangle( CollisionCenter, v1, v2, v3, TriNormal ) )
    {
        // Collision normal is just the triangle normal
        CollisionNormal = TriNormal;
        tMax            = t;

        // Intersecting!
        return true;

    } // End if point within triangle interior

    // Otherwise we need to test each edge
    bCollided |= SphereIntersectLineSegment( Center, Radius, Velocity,
                                             v1, v2, tMax, CollisionNormal );

    bCollided |= SphereIntersectLineSegment( Center, Radius, Velocity,
                                             v2, v3, tMax, CollisionNormal );

    bCollided |= SphereIntersectLineSegment( Center, Radius, Velocity,
                                             v3, v1, tMax, CollisionNormal );

    return bCollided;
}
```

Keep in mind that while at several points in the various intersection techniques we temporarily calculate the actual intersection point, the only thing this function actually returns to the caller is a *t* value and a collision normal. It is the caller that will use this information to return the actual position of the closest intersection back to the CollideEllipsoid function as the new position of the moving entity.

We have spent the last several sections discussing how to detect the intersection between a swept sphere and a triangle. However, we have not yet introduced the function which serves as the glue that binds the

detection phase together. Earlier we covered the code to the CollideEllipsoid function, which was called by the application each time it wanted to update a moving entity. This function would invoke the collision detection phase to determine an intersecting position for the entity and generate a collision normal if an intersection occurred. This would serve as the data for a new sliding plane, upon which we would project our velocity and attempt to shift positions. The function would go back and forth between detection and response until the initial velocity was all used up (via projection onto the slide plane) or until a slide vector was calculated that the sphere could move along and spend the remainder of its velocity without any further intersections. CollideEllipsoid would iteratively make the call to the EllipsoidIntersectScene function, which is the main function in the detection phase. It is this function that will call the SphereIntersectTriangle function for every potential colliding polygon in the collision system's geometry database. It is this function that, after calling the SphereIntersectTriangle function for every triangle, will calculate the updated position for the ellipsoid and return it back to the response phase. Let us look at this function then so that we can get a final view of our overall collision system.

12.10 The EllipsoidIntersectScene Function

This function is the front end for the collision detection phase and is ultimately responsible for returning collision data back for use in the response phase. You will recall from our earlier discussions that it was passed the eSpace position of the center of the ellipsoid, the ellipsoid's radius, and the eSpace velocity vector. Hopefully, you also remember that our CCollision class maintains an array of **CollIntersect** structures which are used to transport collision information back to the response phase. Although the detection phase is only interested in returning the new position of the ellipsoid based on the closest intersection, it may be the case that the sphere intersects several triangles at exactly the same time. When this is the case, the information about each intersection will be returned in this array.

```
struct CollIntersect
{
    D3DXVECTOR3     NewCenter;
    D3DXVECTOR3     IntersectPoint;
    D3DXVECTOR3     IntersectNormal;
    float           Interval;
    ULONG           TriangleIndex;
};
```

The CollIntersect structure stores the new position of the ellipsoid so that the application can update the position of the moving entity. It also stores the actual intersection point on the surface of the ellipsoid and the collision (slide plane) normal. It should be noted that these vectors are all currently in eSpace and should be transformed back into world space before being handed back to the application. We saw that this is exactly what the CollideEllipsoid function did before returning the final position and integration velocity back to the application. The *t* value of intersection is also stored in the Interval float. If multiple collisions occur simultaneously, the information will be returned in multiple CollIntersect structures. When this is the case, keep in mind that the new position of the ellipsoid in eSpace and the interval will be exactly the same in each structure. Since we only return information for the nearest intersection, they must share the same *t* value.

Each structure also returns the index of the triangle in the collision system's geometry database that was intersected. This can be very useful to an application. For example, imagine you were using this collision system to track a ball around a pinball table. When the ball hits certain triangles, our application may want to know which ones they were since this may increase or decrease the player's score. Alternatively, you might decide to check some application associated triangle material property and play a certain sound effect (maybe a different footstep sound depending on the material underfoot) or trigger a scripted event upon a particular collision.

Although we have not focused much on how the collision geometry will be stored in our collision system (this is covered in detail in the workbook), for now just know that we use registration functions such as CCollision::AddIndexedPrimitive to register all the triangles in a mesh with the collision system. In fact, there is even a CCollision::AddActor function which steps though the frame hierarchy of an actor, transforms each mesh into world space, and then adds the triangles of each transformed mesh to the collision database. This means that we could load an entire scene from a hierarchical X file into an actor, and then register all the triangles in that scene with the collision system with a single function call.

Ultimately, each triangle that is registered with the collision system has its vertices added to an internal array of D3DXVECTOR3s. In our case, we are only interested in the positional information at each vertex and there is no need to store other per vertex properties. The CCollision class uses an STL vector to manage this data. The following line of code is located in CCollision.h and defines an STL vector of D3DXVECTOR3 structures called CollVertVector. The vertices of every triangle registered with the collision system will be added to this single array.

```
typedef std::vector<D3DXVECTOR3>    CollVertVector;
```

A vector of this type is then declared as a member variable of the CCollision class as shown below.

```
CollVertVector      m_CollVertices;
```

When the vertices of a triangle are added to the vector, its indices are also stored inside a CollTriangle structure (also defined in CCollision.h) along with other per triangle properties such as its normal and a surface material index which we might find useful. This structure is shown below.

```
struct CollTriangle
{
        ULONG       Indices[3];
        D3DXVECTOR3 Normal;
        USHORT      SurfaceMaterial;
}
```

We called our triangle data a material index since it is common for collision systems to use surface material properties to determine responses (e.g., the application of friction). Of course, you could decide to make this a more generic data storage concept. As discussed a moment ago, you might use this member to store a value that can be added to the player's score or health when a triangle of this type is collided with. Or you might even decide to use this value to index into an array of callback functions. Basically, the idea is to provide us the ability to store a per-triangle code that might be useful in our application.

Every triangle that is added to the collision system is stored in one of these structures. Its three indices reference into the CollVertVector array describing the vertices that comprise the triangle. Triangle storage will take the form of a dynamic array (another STL vector) of CollTriangle structures as shown below. This array is called CollTriVector.

```
typedef std::vector<CollTriangle>   CollTriVector;
```

A vector of this type is declared as a member of the CCollision class as shown below.

```
CollTriVector       m_CollTriangles;
```

It is not important in this discussion to understand how the triangle data gets added to these arrays; the workbook will discuss such application specific topics. However, we do know that our EllipsoidIntersectScene function can access all the triangle and vertex data in these two arrays. Indeed our collision geometry database at this point is basically just these two STL vectors.

Now that we know how the collision data is stored inside the CCollision class, let us have a look at the parameter list to the function.

```
bool CCollision::EllipsoidIntersectScene( const D3DXVECTOR3 &Center,
                                          const D3DXVECTOR3& Radius,
                                          const D3DXVECTOR3& Velocity,
                                          CollIntersect Intersections[],
                                          ULONG &IntersectionCount )
```

The Center parameter and the Velocity parameter describe the center position of the ellipsoid and the direction and magnitude for the movement request. Recall that before this function was called by the CollideEllipsoid function, these values were already in eSpace. We also pass in the radius vector of our ellipsoid, which is used to convert each triangle into eSpace prior to testing it for intersection. As the fourth parameter, the CCollision object's CollIntersect array is passed so that the detection function can fill it with intersection information and return it to the response phase. The final parameter is passed by reference from the CollideEllipsoid function. It is a DWORD value which will keep track of how many CollIntersect structures in the passed array contain valid intersection information. Remember, this array is re-used to transport information every time this function is called.

> **Note:** If necessary, please refer back to our discussion of the CollideEllipsoid function. This is the function that calls EllipsoidIntersectScene and retrieves the information from the CollIntersect array.

Let us now look at this final function one section at a time. It is responsible for testing each triangle in the database for intersection and recording the nearest intersection results in the input **CollIntersect** array.

The first part of the function calculates the inverse radius vector of the ellipsoid. This will be needed to scale the vertex of each triangle into eSpace. We also copy the passed parameters Center and Velocity into two local variables called eCenter and eVelocity.

```
bool CCollision::EllipsoidIntersectScene( const D3DXVECTOR3 &Center,
                                          const D3DXVECTOR3& Radius,
                                          const D3DXVECTOR3& Velocity,
                                          CollIntersect Intersections[],
                                          ULONG & IntersectionCount )
{
    D3DXVECTOR3 eCenter, eVelocity, InvRadius;
    D3DXVECTOR3 ePoints[3], eNormal;
    D3DXVECTOR3 eIntersectNormal, eNewCenter;
    ULONG       Counter, NewIndex, FirstIndex;
    bool        AddToList;
    float       eInterval;
    ULONG       i;

    // Calculate the reciprocal radius to minimize division ops
    InvRadius = D3DXVECTOR3( 1.0f / Radius.x, 1.0f / Radius.y, 1.0f / Radius.z );

    eCenter   = Center;
    eVelocity = Velocity;

    // Reset ellipsoid space interval to maximum
    eInterval = 1.0f;

    // Reset initial intersection count to 0 to save the caller having to do this.
    IntersectionCount = 0;
```

Notice above that the local variable eInterval is initially set to one. It is this variable that we will be passing into the intersection routines as tMax. As closer intersections are found, the value of eInterval will be overwritten to contain the nearer *t* values. Initializing it to 1.0 describes an intersection time at the very end of the ray. That is, the ray is completely clear from obstruction unless our intersection routines detect a collision somewhere along the ray and update it with a smaller value. We also set the passed IntersectCount variable to zero because we have not yet found any intersections and thus, have not added anything useful to the passed CollIntersect array.

Now we have to loop through each triangle in the database. We create an STL iterator to step though the CollTriVector one triangle at a time. Then we use the triangle's indices array to fetch the correct vertices from the m_CollVertices array. We scale the vertex positions (using theVec3VecScale function) by the Inverse ellipsoid radius vector, transforming them into eSpace, and store them in the local vector array ePoints.

```
    // Iterate through our triangle database
    CollTriVector::iterator Iterator = m_CollTriangles.begin();
    for ( Counter = 0; Iterator != m_CollTriangles.end(); ++Iterator, ++Counter )
    {
        // Get the triangle descriptor
        CollTriangle * pTriangle = &(*Iterator);
        if ( !pTriangle ) continue;

            // Get points and transform into ellipsoid space
            ePoints[0] = Vec3VecScale( m_CollVertices[ pTriangle->Indices[0] ],
                                       InvRadius );
            ePoints[1] = Vec3VecScale( m_CollVertices[ pTriangle->Indices[1] ],
                                       InvRadius );
```

```
            ePoints[2] = Vec3VecScale( m_CollVertices[ pTriangle->Indices[2] ],
                                       InvRadius );

            // Transform normal and normalize
            // (Note how we do not use InvRadius for the normal)
            eNormal = Vec3VecScale( pTriangle->Normal, Radius );
            D3DXVec3Normalize( &eNormal, &eNormal );
```

We convert the normal of the current triangle into eSpace as well, by scaling it with the radius of the ellipsoid and normalizing the result. Note that we do not use the inverse radius as is the case with the vertices; it works the other way around for direction vectors.

At this point we have the three eSpace vertices for the current triangle we are about to test and also have its eSpace normal. We now have everything we need to call our very familiar SphereIntersectTriangle function:

```
        // Test for intersection with a unit sphere
        // and the ellipsoid space triangle
        if ( SphereIntersectTriangle( eCenter, 1.0f, eVelocity,
                                      ePoints[0], ePoints[1], ePoints[2], eNormal,
                                      eInterval, eIntersectNormal ) )
```

When we call this function we pass in the eSpace center of the ellipsoid (the ray origin) and the radius of the ellipsoid in eSpace. We know that this is always 1.0 since the ellipsoid in eSpace becomes a unit sphere. We also pass in the eSpace velocity vector (ray delta vector) followed by the three eSpace vertices of the triangle. The eighth parameter is the eInterval local variable which was set to 1.0 at the start of the function. If an intersection occurs between the swept sphere and this triangle, this variable will contain the new (smaller) *t* value on function return. As the final parameter we also pass in a local 3D vector variable (eIntersectNormal) which will store the collision normal if the function returns true for intersection.

The next section of code is inside the conditional code block that is executed if an intersection occurs (i.e., SphereIntersectTriangle returns true). Let us look at this code a few sections at a time.

The first thing we do is test the value currently stored in the local eInterval variable. If we are in this code block, then SphereIntersectTriangle not only determined an intersection, but it also found one that occurs closer to the ray origin then any previous found. Since eInterval is passed into this function for *t* value storage, the value stored in the variable at this point will be the new *t* value of intersection along the velocity vector.

If this is a positive value, then eInterval contains the parametric distance of the intersection point along the ray and will be in the 0.0 to 1.0 range. When this is the case, the new sphere position can be calculated by scaling the velocity vector by this amount and adding it to the current center position of the sphere. If eInterval is a negative value then it means the sphere was embedded in the triangle. When this is the case, eInterval will store the actual world space distance by which the sphere's center must be moved back in the direction of the triangle plane normal. This will assure that the surface of the sphere is no longer embedded.

```
    {
            // Calculate our new sphere center at the point of intersection
            if ( eInterval > 0 )
                eNewCenter = eCenter + (eVelocity * eInterval);
            else
                eNewCenter = eCenter - (eIntersectNormal * eInterval);
```

At this point we now have the new eSpace position of the sphere. Of course, we have many triangles to test, so we may find closer intersections and overwrite the new sphere center position with an even closer one in future iterations of the loop.

As discussed, the intersection information will be returned from this function using the Intersections array that was passed in. At first the IntersectCount value will be zero and we will store this new intersection information at the start of the array. If we find that our new intersection *t* value is smaller than the information we currently have stored in the Intersections array, we will overwrite it with the new information. Remember, all we wish to return in the Intersections array is the closest collision. If intersection happens simultaneously with multiple triangles, this will require us to store the information for each of these collisions in its own CollIntersect structure.

If we do find an intersection which element in the array should we store it in? We first test to see if IntersectionCount equals zero. If it does, then this is the first intersection we have found and therefore, we will store this information at index zero in the Intersections array. If the interval is smaller than the interval currently stored in the first element in the array, it means that our new one is closer and we should discard all the others and add this new one in their place. To do this we simply say that if the new intersection interval is smaller than the intervals currently stored in the intersection array, set the index to 0 so that it overwrites the first element in the array. Then set the number of intersections to 1 so all older information (that may still be stored in elements 2, 3, and 4, etc.) are ignored.

```
            // Where in the array should it go?
            AddToList = false;
            if ( IntersectionCount == 0 || eInterval < Intersections[0].Interval )
            {
                // We either have nothing in the array yet,
                // or the new intersection is closer to us
                AddToList         = true;
                NewIndex          = 0;
                IntersectionCount = 1;
            } // End if overwrite existing intersections
```

The else code block associated with the conditional above is executed only if the new interval returned for this latest intersection matches (with tolerance) the interval values currently stored in the intersection array. Essentially we are saying that if our intersection interval is the same as the intervals we currently have stored in our intersection array, then we have found another collision that happens at the exact same time as those already in the list. Thus, the size of the array should be increased and the new data added to the end.

```
            else if ( fabsf( eInterval - Intersections[0].Interval ) < 1e-5f )
            {
```

```
                // It has the same interval as those in our list already, append to
                // the end unless we've already reached our limit
                if ( IntersectionCount < m_nMaxIntersections )
                {
                    AddToList         = true;
                    NewIndex          = IntersectionCount;
                    IntersectionCount++;

                } // End if we have room to store more

            } // End if the same interval
```

Studying the two code blocks above you should be able to see that if the intersect array contained 10 CollIntersect structures describing collisions that happen at time $t = 0.25$, as soon as a intersection happens with a t value of 0.15 (for example), the structure at the head of the array would have its data overwritten and the array would be snapped back to an effective size of 1 element.

Provided one of the above cases was true, the local AddToList Boolean will be set to true. Checking that this is the case, we then finally add the collision data to the intersection array. NewIndex was calculated in the above code blocks and describes whether we are adding the data to the back of the array, or whether we are overwriting the head of the array.

```
            // Add to the list?
            if ( AddToList )
            {
                Intersections[ NewIndex ].Interval  = eInterval;
                Intersections[ NewIndex ].NewCenter = eNewCenter +
                                                      (eIntersectNormal * 1e-3f);
                Intersections[ NewIndex ].IntersectPoint = eNewCenter –
                                                           eIntersectNormal;
                Intersections[ NewIndex ].IntersectNormal = eIntersectNormal;
                Intersections[ NewIndex ].TriangleIndex = Counter;

            } // End if we are inserting in our list

        } // End if collided

    } // Next Triangle

    // Return hit.
    return (IntersectionCount > 0);
}
```

We do this for every triangle and finally return true or false depending on whether at least one intersection was found (i.e., IntersectionCount > 0).

Notice in the above code how interval, the collision normal (eIntersectNormal) and the triangle index (Counter) are copied straight into the relative fields of the CollIntersect structure. Also note how the new non-intersecting position of the center of the sphere (moved as far along the velocity vector as possible) is slightly pushed back from the plane by a distance of 0.0001. This is something we added to provide a little more stability when working in the world of floating point inaccuracies. Although we know that our intersection functions have correctly determined an interval value that was used to create the new

position of the center of the ellipsoid, it is positioned such that its surface is just contacting the triangle. We do not want floating point rounding errors to turn this into an embedded situation, so we slightly shift the sphere away from the triangle by a very small amount to give us a little bit of space between the surface and the ellipsoid.

Finally, notice that we return the intersection position (in eSpace). If you refer back to our discussion about how the slide plane normal is generated, you will see that moving the new sphere position along the slide plane normal (which is unit length and also equal to the radius of the unit sphere) will provide us with the intersection point in eSpace on the surface of the unit sphere. Although our particular response implementation may not require all of this data, it seemed better to provide as much information about the collision as we can and let the caller choose what they need to accomplish their specific objectives.

Believe it or not, at this point, we have now finished our core collision detection and response system implementation discussion. Moving forward, we now have a robust way to collide and slide our entities as they intersect with static scene geometry. This will certainly make our demos feel much more 'game-like' than before.

12.11 Final Notes on Basic Detection/Response

We have obviously discussed a good deal of code in this textbook, but we have tried to keep that discussion as tied into the theory and the mathematics as possible. Utility functions that add polygons to the collision system's database and such minor players have not been covered here. These will fall into the domain of the chapter workbook and as such, a more complete description of the source code can be found there.

We have now learned how to implement a collision detection and response system that works nicely with moving entities and static scene geometry. However, some of you may have crossed referenced the code shown in this chapter with the code in Lab Project 12.1 and encountered some differences. For example, while the core intersection routines are the same, the CollideEllipsoid function and the EllipsoidIntersectScene functions probably look a bit more complicated in Lab Project 12.1. Of course, there is a perfectly good reason for this -- in Lab Project 12.1, we have implemented a more fully featured system that supports terrains, actors, and even referenced actors. In other words, the collision system supports animated frame hierarchies. This means that we could register an actor with the collision system where that actor has moving doors, lifts, and conveyor belts that are updated every time the actor's AdvanceTime function is called. Our collision system will track the movement of these dynamic objects and make sure that the entity that is having its position updated correctly intersects and responds to this dynamic geometry. For example, if the player stands on an animated lift platform, when the lift moves upwards, the player's ellipsoid should also be pushed upwards with it. Doors that swing or slide open should correctly knock the player back, and so on.

This is quite a complex system and its code could not possibly be tackled until the foundation of a more simplistic collision detection and response phase was put into place. Therefore, the majority of this chapter has been focused on presenting a system that deals only with static geometry. That is, once the

mesh/actor/terrain/etc. triangles are registered with the collision system's database in world space, that is how they will remain. This works perfectly for many situations and definitely has allowed us to come to grips with implementing a 'collide and slide' system.

Now the time has come to take the next step along this road and look to include support for dynamic collision geometry. All of the work we have done so far has not been in vain; supporting dynamic objects will involve adding features to the code we currently have, not replacing it. Our discussions of adding dynamic object management to our collision system will be from a relatively high level here in the textbook and we will save the detailed source code examination for the workbook.

12.12 Dynamic Collision Geometry

Our collision system currently stores its geometry using two dynamic arrays. Every single triangle added to the system will have its vertices stored in the CCollision::m_CollVertices array and its triangle information (indices, normal, etc.) stored in the CCollision::m_CollTriangles array. As we have seen, it is these two arrays that contain all the geometry that we wish to be considered as collidable with respect to our moving objects. We will see in the workbook that when we add a mesh to the collision system, all its vertices and indices will be added to these arrays.

We can add an entire CActor to the collision system also. The CCollision::AddActor function is passed an actor and will automatically traverse the frame hierarchy looking for mesh containers. For each mesh container it finds, it will grab its vertices and indices and add them to these two arrays. This single function could be used to register your entire scene with a single function call were it all to be stored in a single frame hierarchy loaded into a CActor object. We even have functions to register our CTerrain objects with the collision database.

All of these functions are simple helper functions that will take the passed object (CActor, CTerrain, etc.) and collect its polygons, transform them into world space, and add them to the two collision geometry arrays. This means we can integrate our collision system into our games with only a few simple calls to such registration functions. Once we have done this for each object we wish to register, our CCollision geometry database (m_CollVertices and m_CollTriangles) will contain every triangle we need to test against whenever a moving entity is updated. As we have seen, it is these two arrays that are looped through in the EllipsoidIntersectScene function. One at a time, each triangle is fed into the SphereIntersectTriangle function to determine if a collision occurred.

So far, so good. But the collision database currently has no capability to manage dynamic scene objects. Earlier in the course we spent a good deal of time covering the D3DX animation system and our CActor class fully supports it. If we register an actor with our collision system and that actor contains animation data, if we use its animation functions after we have registered the actor with the current collision database, we will be in for some real trouble. After all, as we just mentioned, those triangles are going to be converted to world space during collision registration and remain static.

Consider a scene that models the interior of a building. Animations might be triggered that make doors open and close, lifts go up and down, etc. As soon as an animation is played that moves the position of

any of these meshes, the triangle data that the collision system has in its static arrays will be out of date and will no longer represent what the scene actually looks like to the player. Keep in mind that our collision and render geometry are separate things. So a door that opens for example, would visually appear to present a new path for the player to explore, but if it was registered with the collision system in an initially closed state, it would remain so and prevent the player from crossing the threshold (and vice versa if the door was registered in an open state).

Clearly, we need some way of letting the collision system know that some of its geometry is dynamic. As such, a new mechanism will have to be added to our collision system for such objects. Processing dynamic scene objects will place an even greater burden on our CPU (more so than testing collisions against our static geometry arrays), so we should only use the dynamic object system for meshes or hierarchies that we intend to animate. Geometry that will never change should still be added to the collision system as before for storage in the static arrays. What we will add is a secondary sub-system that will maintain a list of dynamic objects which the collision system needs to be kept informed about. This way it can track positional and/or rotational changes.

After reading this section, you should go straight to the workbook where we will finally implement everything we have discussed. This section of the textbook will focus primarily on the type of system design we need to think about and also discuss how to sweep a sphere against moving scene objects.

12.12.1 Sweeping a Sphere against Dynamic Objects

A dynamic object's data structure will contain the model space vertex and index data and a pointer to the object's world matrix. Because this is a matrix pointer, the collision system will always have access to the current object matrix, even when its contents are modified from outside the collision system by the application.

Our EllipsoidIntersectScene function must additionally perform the sphere/triangle tests against the geometry of each dynamic object. Nothing much has changed with respect to our intentions in the detection step -- we are still searching for the closest intersecting triangle. Whether that triangle exists in the static triangle array or in the triangle array of the one of the dynamic objects is for the most part irrelevant. What is quite different however is how we sweep the sphere against the dynamic objects. We can no longer just sweep the sphere as we did previously since the dynamic object is theoretically moving as well.

It would seem that we have to sweep the sphere along its velocity vector and the dynamic object along its velocity vector and then determine the intersection of the two swept volumes. Indeed that is one possible approach, but it is made complicated by the fact that the shape of the dynamic object might be something as simple as a cube or as complex as an automobile mesh. We know that we can sweep a sphere/ellipsoid easily (which is why it was used to bound our moving entity), but we hesitate to use crude approximations like this when sweeping the dynamic scene objects since we are after exact triangle intersections, not just the intersections of two swept bounding volumes.

Our system will take a different view of the problem. In fact, we will not have to sweep the dynamic scene object at all. What we will do is cheat a little bit by taking its movement into account and extending the swept sphere to compensate. That is, for each dynamic object we will maintain its current world matrix and its previous world matrix. Every time the application updates the position or orientation of a registered dynamic object, it will inform the collision system and the system will copy the currently stored matrix into the previous matrix variable. These two matrices will tell the collision system where the object is now and where it was in the previous collision update. By inverting the previous matrix and multiplying it with the current matrix we essentially end up with a relative movement matrix (we call this a *velocity matrix* in our system). This matrix describes just the change in position and orientation from the previous collision update to the current one for the dynamic object. We can use this information to grow the size of our swept sphere so that it compensates for how much the dynamic object has moved since the previous update. Since we have grown our sphere by the amount the object has moved from its previous position, we can now sweep it against the triangles of the dynamic scene object in its previous position. This once again reduces our intersection testing to a simple sphere/triangle test for each triangle in the dynamic object, and we already have functions written to do that.

Let us have a look at some diagrams to clarify this point. In Figure 12.40 we see our sphere about to be moved along its velocity vector. As we know, our collision system will use this information to create a swept sphere which can then be intersected against each of the triangles in the scene database. In Figure 12.40, we also see a blue cube, which we will assume is a dynamic object, which is currently stationary.

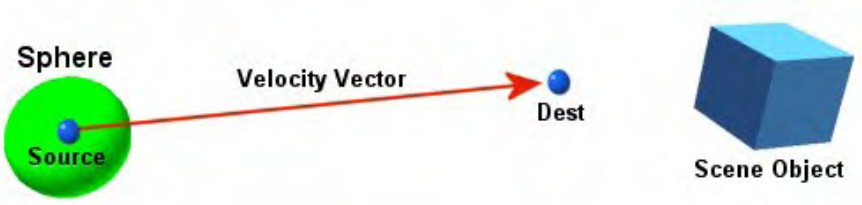


Figure 12.40

When this is the case, our job is no different then detecting collisions between our swept sphere and any other object that has been added to the static geometry database.

We know that if the dynamic object is not currently being treated as dynamic, then we can just use the SphereIntersectTriangle function to sweep the sphere against each triangle in its triangle array. Remember, each dynamic object will store its own array of triangles and vertices and therefore, even when not currently in motion, our collision system will have to loop through each dynamic object and test its triangles against the swept sphere. This is in addition to the triangle tests performed on the static geometry array. At the end of the day, we are just trying to find the closest intersecting triangle so that we can adjust the sphere's position so that it only moves along the velocity vector up to the time the sphere is in contact with the surface. This position and the corresponding slide plane normal are then returned back to the collision response phase.

In Figure 12.41, the sphere will not intersect the cube while moving along its current velocity vector. We know that in this case, the final position returned to the application for the

Figure 12.41

ellipsoid will be the original one that was requested (Sphere Center + Velocity Vector). This is shown as Dest in Figure 12.40.

Complications arise as soon as we start moving these dynamic objects around in our application. As discussed, each dynamic object will also maintain a pointer to the matrix that the application is using to position it in world space. This means our collision system always has access to the current position of any of its dynamic objects. Figure 12.42 depicts the problem we need to solve when the dynamic object is moving. In this example, the position the sphere wants to move to bypasses the position the dynamic object wants to move to. As such, a head on collision would occur somewhere along their respective paths.

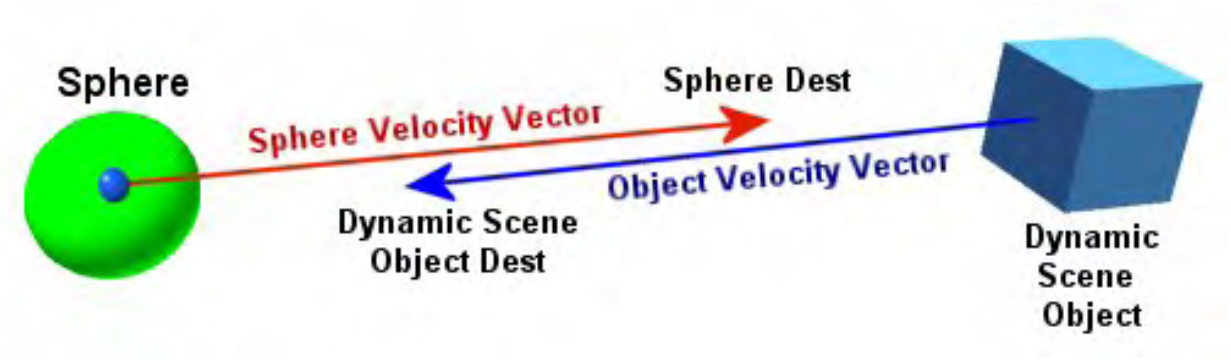


Figure 12.42

We can see clearly that there is a section of overlap between the two projected movement paths and a collision is definitely going to happen. The question is, what should our response be? Should the sphere and the cube stop at the point of intersection? Should they both bounce backwards as a result of impact?

With some caveats, in our implementation, we have decided to provide the dynamic scene objects with movement precedence. Since our collision system only has control over positioning the sphere, it was simplest just to give the dynamic scene object total right of way. However, this does not mean a user cannot implement complex responses, like allowing for various principles of physics to come into play. Our system does have a pointer to the object's world matrix and as such, it could overwrite the values in that matrix with anything it pleases. Of course, while this is certainly true, we would have to proceed very carefully.

Imagine for example a situation where we have registered an actor as a dynamic object with our collision system. The CCollision::AddActor function will have a Boolean parameter used to indicate whether we would like the meshes in the actor's frame hierarchy to be added to the collision system's static database or whether we would like to add each mesh in the hierarchy as a dynamic object. Obviously, if we intend to play animations on our actor, we will want it registered as set of dynamic objects (assuming multiple meshes). Now imagine that our cube in Figure 12.42 was one of many dynamic objects that were created from that single actor. The matrix pointer that the collision detection system stored would point directly at the combined frame matrix for the mesh. When an animation is played, since those animations are pre-recorded, we cannot simply move the cube backwards when an impact occurs. Even if our collision system were to alter the matrix of the dynamic object, the values it placed in there would be overwritten in the next frame update when the application advanced the animation time and the absolute frame matrices were all rebuilt. Therefore, only for a split second would

these matrices, stored external to the collision system, contain the values placed in them by the collision system. As you can see, at least in this type of situation, we are going to have to forget about updating the matrices of the dynamic objects and update the position of the sphere instead.

This is not necessarily such a bad thing. Usually we want the animations that were recorded to play out the same way regardless of where the player may be. For example, we might load a scene which has animations that control large pieces of machinery in some part of the level. If the player was to walk into a swinging piece of the machinery, they should probably be knocked out of the way, leaving the machinery to carry out its animation unimpeded. This is also generally true when dealing with lifts. When the player is standing on the lift and animation is played which raises the lift up, we are relying on this behavior to make sure that the lift always moves the player up and does not bounce off the feet of the player when the two collide. Therefore, when the sphere collides with a moving object, we will move the sphere back so that its new position is simply contacting the object. It might even be the case that the new center position of the sphere is actually behind its original starting position if the sphere has been pushed backwards by a moving scene object (given some amount of force that might be applied).

To be fair, this approach is not always going to be the ideal way to handle every single scenario you and your design team might come up with. For very heavy objects like the machinery we just mentioned, this probably works well enough, but for objects with little mass or momentum, you might decide to go ahead and actually work out a way to modify the animation on the fly or take some other action. For example, you might decide to add a mechanism to the system which allows you to pass in a callback function at registration time. That callback could be triggered by the system when an object collides with the dynamic object. The callback might be passed information about the colliders such as their masses, linear and angular velocities, etc. You could then make some decisions based on the relative physical differences between the two colliders. If the dynamic object was a ceiling fan for example, you might decide to immediately stop the pre-recorded animation and take no action on the moving entity. Different situations will require different responses, but you get the idea. In our case, we will simply allow the dynamic objects to proceed with their animations and limit our response results to the moving entity only. This will make things much easier to implement and understand in the short tem.

Figure 12.43 illustrates this concept using the previous example. In Figure 12.42 we saw how the sphere and the cube wanted to move along paths that would cause a collision. In Figure 12.43 we see the result of such an impact. The cube is allowed to move from its previous position into its new position and the sphere is moved back so that it is now resting against the new position of the cube.

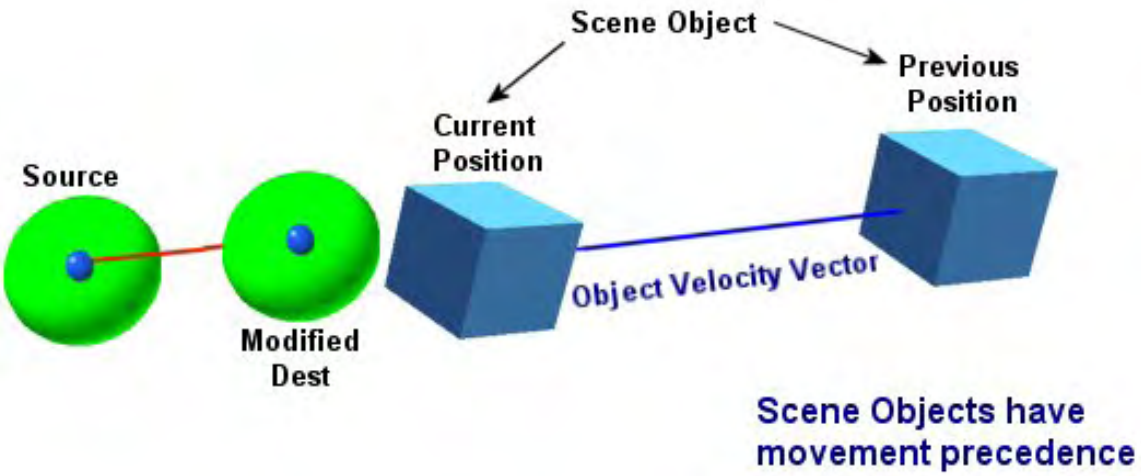


Figure 12.43

So we know how we want our ellipsoid to intersect with moving objects. But how do we find the final position for the sphere in the above situation? As discussed, the cube will have its matrix updated by the application. So what we have is the new cube position and the velocity we wish to move along. Now, you might assume that all we have to do it sweep the sphere against the cube in its current position. This would certainly seem to work in the above diagram and it would also mean that the test is once again reduced to a static test between the swept sphere and the triangles of the cube in its new position. However, imagine a situation where the cube was moving so fast, that while its start position was in front of the swept sphere (as in Figure 12.43), its final position is actually behind the swept sphere. In other words, with a single matrix update, the dynamic object has leapt from one side of the swept sphere to the other. Neither the previous or current positions of the cube intersect the swept volume, so our collision system would determine that the sphere could move to its desired location. The problem is that between these two frame updates, the cube has just passed straight through our sphere with no ill effect. So what should have happened instead? The sphere should have been knocked backwards by the moving cube (via some amount of applied force).

We discussed previously that a dynamic object will contain a pointer to its current application controlled matrix and we will cache its previous matrix as well. When we add a dynamic object to our collision system, a new DynamicObject structure will be allocated to contain this information along with some other data. If an actor is registered with the collision system, a DynamicObject structure will be created for every mesh in its hierarchy. Our DynamicObject structure will look like this:

Excerpt from CCollision.h

```
struct DynamicObject
{
    CollTriVector   *pCollTriangles;
    CollVertVector  *pCollVertices;
    D3DXMATRIX      *pCurrentMatrix;
    D3DXMATRIX       LastMatrix;
    D3DXMATRIX       VelocityMatrix;
    D3DXMATRIX       CollisionMatrix;
    long             ObjectSetIndex;
    bool             IsReference;
};
```

CollTriVector *pCollTriangles

This is a pointer to an STL vector of CollTriangle structures. This vector contains one entry for each triangle in the mesh of the dynamic object. A CollTriangle contains the indices into the pCollVertices vector and the triangle normal.

CollVertVector *pCollVertices

This is a pointer to an STL vector of D3DXVECTOR3 structures. This vector will contain all of the *model space* vertex positions for the dynamic object's mesh.

D3DXMATRIX *pCurrentMatrix

When a dynamic object is added to the collision system via the CCollision::AddDynamicObject method, we must also pass in the address of the dynamic object's world matrix. This matrix will be owned and altered by the application. A pointer to this matrix is stored in the pCurrentMatrix member so that the collision system can always have immediate access to the matrix and any changes the application may have made to it. Our collision system will never write any values to this matrix; it will instead use it only to copy the values from this matrix into the dynamic object's LastMatrix member when the collision system is notified that the matrix has changed. If the dynamic object is part of a frame hierarchy, this member will point to the associated frame's combined matrix.

D3DXMATRIX LastMatrix

Every time the application alters the position of an object, it has to inform the collision system that the object has been updated and its matrices need recalculating inside the collision system. This is done via the CCollision::ObjectSetUpdated method. The pCurrentMatrix pointer will always point to the matrix of the object with the updated values, so we will use this matrix to copy the values into the LastMatrix member. Thus, the LastMatrix member will always contain the current world matrix of the object during collision tests.

D3DXMATRIX CollisionMatrix

When the ObjectSetUpdated function is called, the matrix that was previously the current position/orientation of the object is stored in LastMatrix. This is now about to be overwritten by the new values in the matrix pointed to by the pCurrentMatrix pointer. Before we do that, we copy the previous matrix into the CollisionMatrix member so that at the end of the update, LastMatrix contains the new current position of the object and CollisionMatrix holds the previous position of the object. The reason it is called the collision matrix and not something like 'PreviousMatrix' is something that will become clear shortly.

D3DXMATRIX VelocityMatrix

The velocity matrix is calculated for the dynamic object every time its position is updated by the application and the ObjectSetUpdated function is called. We need this matrix to describe the direction and magnitude the object has travelled from its previous position since the last collision frame update. That is to say, this matrix will describe the current position/orientation of the object relative to the previous position/orientation currently stored in Collision Matrix. How do we calculate it? We take the previous object world matrix and invert it. We then multiply this with the current object world matrix. Because we inverted the previous world matrix prior to the multiply, this has the effect of subtracting the position and orientation stored in the previous matrix from the position and orientation stored in the current matrix. The result is our velocity matrix which describes the new position and orientation of the

object relative to its previous one. You will see in a moment how we will use this matrix to stretch the swept sphere to compensate for the movement of the dynamic object from its previous position. This will then allow us to sweep this extended sphere against the triangles of the dynamic object in their previous positions (described in CollisionMatrix) using the usual static sphere/triangle intersection code we developed for our static scene geometry tests.

long ObjectSetIndex

Every time a dynamic object is registered with the collision system it will return an ObjectSetIndex. This is just a numeric value that uniquely identifies this object to the system. You can think of it as a way for the application to tell the collision system which object had its matrix updated. When adding a single dynamic object to the system, the dynamic object will be added to the collision system's dynamic object array and its internal dynamic object counter incremented. This will be returned as the object set index for the newly added object. The application should store this away so that it can always identify which object needs to be updated in the collision system. What we do in Lab Project 12.1 is add this as a new member in our CObject structure, which we have been using in one shape or form since the beginning of this course series. The updated CObject structure is shown below.

```
class CObject
{
public:
                CObject( CTriMesh * pMesh  );
                CObject( CActor   * pActor );
                CObject( );
    virtual     ~CObject( );

    D3DXMATRIX                m_mtxWorld;
    CTriMesh                 *m_pMesh;
    CActor                   *m_pActor;
    LPD3DXANIMATIONCONTROLLER m_pAnimController;
    long                      m_nObjectSetIndex;
};
```

Notice that even if the object contains an actor comprised of many meshes, only one object set ID is assigned by the collision system for all the meshes contain within. This is by design. When an actor is registered with the collision system, every mesh in the hierarchy will have a dynamic object created for it and added to the collision system's dynamic object database. However, every object created from the hierarchy will also be assigned the same object set index. This ID is used to identify a set of objects that have some relationship. To see why this is necessary, imagine the situation where you have registered an actor containing 50 meshes with the collision system. Then imagine that the application moves the entire actor to a new position in the world. When the actor's hierarchy is updated, all the absolute matrices in the hierarchy will be rebuilt to describe new world space positions of all 50 meshes contained within. The 50 dynamic objects in the collision database that we created from that hierarchy have just all had their current matrices changed. Because we have assigned every dynamic object in that hierarchy the same ID, we can call the ObjectSetUpdated function, passing in the actor's ID, and it will re-cache the matrices for every dynamic object with a matching ID (i.e., the 50 dynamic objects that were originally created from the hierarchy we just updated). Thus the object set ID does not just describe the unique ID of a single dynamic object, it can also represent a family of dynamic objects that will always need to be updated together.

bool IsReference

The final member of the DynamicObject structure is a boolean that indicates whether or not this dynamic object should be a reference. Referencing will be discussed in the workbook, but essentially this just provides a way for multiple dynamic objects to share the same triangle and vertex data.

We have now covered all the members of the DynamicObject structure. To understand the collision system, we first should understand how the application informs the collision system that a dynamic object has been updated. We will assume at this point that the objects have already been registered with the collision system and have been assigned object set index numbers. The registration functions are fairly simple and as such will be covered in the workbook. All we are interested in discussing here is how the dynamic system is going to work from a high level.

The CScene::AnimateObjects function is another familiar method. It is called from CGameApp::FrameAdvance function to give the scene class a chance to update the positions of any objects it wishes to animate. Since learning about animation, we have always used this function to advance the timer of each actor being used by the scene. Below we see the first part of the function from Lab Project 12.1. There is nothing new in this first section; it loops though each CObject and tests to see if an actor lives there. If not, then there is nothing to animate so it continues to the next iteration of the loop. If an actor does live there, then it attaches the CObject's animation controller to the actor.

Excerpt from CScene.h

```
void CScene::AnimateObjects( CTimer & Timer )
{
    ULONG i;

    // Process each object for coll det
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CObject * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Get the actor pointer
        CActor * pActor = pObject->m_pActor;
        if ( !pActor ) continue;

        // If we are instancing, make sure that the cloned controller
        // is re-attached (no need to resynchronize the animation outputs
        // as we're about to call 'AdvanceTime'.)
        if ( pObject->m_pAnimController )
             pActor->AttachController( pObject->m_pAnimController, false );

        // Advance time
        pActor->AdvanceTime( Timer.GetTimeElapsed(), false );
```

The next section is new.

If the object has a value in the m_nObjectSetIndex member of -1, then it means this object has not been registered with the collision system and we can simply continue on to process the next object. If this is not the case and we have a valid collision ID stored in m_nObjectSetIndex, then we know that the actor we are animating is also a dynamic object in the collision system and the system needs to be informed of

the changes we have made to its position/orientation. We first set the actor world matrix so that all its absolute frame matrices get built, and then we call the CCollision::ObjectSetUpdated to inform the collision system about the event. The collision system can then grab the current state of the matrices for each dynamic object created from this hierarchy and calculate the collision and velocity matrices.

```
        // If the object has a collision object set index,
        // update the collision system.
        if ( pObject->m_nObjectSetIndex > -1 )
        {
            // Set world matrix and update combined frame matrices.
            pActor->SetWorldMatrix( &pObject->m_mtxWorld, true );

            // Notify the collision system that this set of dynamic objects
            // positions, orientations or scale have been updated.
            m_Collision.ObjectSetUpdated( pObject->m_nObjectSetIndex );

        } // End if actor exists

    } // Next Object

}
```

Let us now take a look at the CCollision::ObjectSetUpdated function. Although we do yet understand how the collision detection is performed, looking at this function will show us how the collision matrix and velocity matrix are created every time the object is updated. It is important that we understand this process since these two matrices will be used by our EllipsoidIntersectScene function when sweeping the sphere against the triangles of dynamic objects.

This function is very small and it has a simple task to perform. It is passed an object set index which it will use to locate dynamic objects in the collision database that share the same ID. If you look in CCollision.h you will see that the CCollision class stores its dynamic object structures in an STL vector.

```
typedef std::vector<DynamicObject*> DynamicObjectVector;
DynamicObjectVector m_DynamicObjects;
```

As you can see, the member variable m_DynamicObjects is an STL vector that stores DynamicObject structures. Let us now have a look at the function itself.

Excerpt from CCollision.cpp

```
void CCollision::ObjectSetUpdated( long Index )
{
    D3DXMATRIX mtxInv;

    // Update dynamic object matrices
    DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
    for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
    {
        DynamicObject * pObject = *ObjIterator;

        // Skip if this doesn't belong to the requested set.
        if ( pObject->ObjectSetIndex != Index ) continue;
```

In this first section of code we create an iterator to allow us to loop through and fetch each DynamicObject structure from the STL vector. We store a pointer to the dynamic object in the pObject local variable for each of access. We then check to see of this object's ObjectSetIndex is the same as the index passed into the function. If it is not, then it means the current object does not belong to the family of objects we are updating, so we continue on to test the rest of the list.

This next section of code is only executed if we have determined that the index of the current object matches the index passed in by the application. If this is the case, then the application has updated the matrix of the dynamic object and we need to do some matrix recalculation inside the collision system.

The first thing we do is access the matrix that is currently stored in the dynamic object's LastMatrix member. This matrix describes the world matrix of the object before the application recently updated it. Inverting it gives us a matrix that essentially subtracts this transformation from any matrix. Thus, if we multiply it with the dynamic object's current matrix, we essentially subtract the old position and orientation from the new one to leave us with a matrix that describes only the relative motion that has taken place since the last update. Remember, pCurrentMatrix is a pointer to the object's actual matrix and we have immediate access to it. We store the relative movement result in the VelocityMatrix member.

```
        // Generate the inverse of the previous frame's matrix
        D3DXMatrixInverse( &mtxInv, NULL, &pObject->LastMatrix );

        // 'Subtract' the last matrix from the current to give us the difference
        D3DXMatrixMultiply( &pObject->VelocityMatrix, &mtxInv,
                            pObject->pCurrentMatrix );
```

Next we copy over the previous matrix of the object from the LastMatrix member into the Collision Matrix member before we update the LastMatrix member with the new current position of the object. We now have everything we need. We have the velocity matrix calculated and the previous world matrix stored in CollisionMatrix which describes the world space position of the object that we will collide against. Further, we have updated LastMatrix with the current matrix so that this whole pattern can repeat again the next time UpdateObjectSet is called.

```
        // Store the collision space matrix
        pObject->CollisionMatrix = pObject->LastMatrix;

        // Update last matrix
        pObject->LastMatrix = *pObject->pCurrentMatrix;

    } // Next Object
}
```

Once again, please do not concern yourself at the moment with how the dynamic objects are created and how the scene registers the dynamic objects with the collision database. This is all pretty routine code which will be discussed in detail in the workbook. For now, all you need to know is that when our application calls the CollideEllipsoid method to move an object about the game world, the collision system will have a velocity matrix describing its movement from its previous position, and a matrix that describes this previous position (CollisionMatrix). This will be stored for every dynamic object.

In the EllipsoidIntersectScene function we now know that in addition to testing the swept sphere against the static scene database, we also have to test for collisions against the triangles of each dynamic object. We have at our disposal the previous transform state of each dynamic object (CollisionMatrix) and the velocity matrix that describes how much that state has changed since the last collision test. We now have to use this information to determine if the dynamic object intersected the swept sphere during its state change. If so, the position of the sphere will need to be adjusted to account for this collision.

Take a look at Figure 12.44. Source and Dest represent the amount the user wishes to move the ellipsoid (in eSpace). The red arrow represents the velocity vector for the ellipsoid (in eSpace) describing the distance and direction we wish to travel. Before testing each triangle of a dynamic object for collision, we will first transform it by the collision matrix so that the triangle is placed in the world at its previous position. Since this is a per-triangle process, we will describe it with respect to a single triangle for now.

The solid green line illustrates the current triangle to be tested from the dynamic object. It is transformed into its previous position. The green dashed line shows where the triangle is in its current position. The blue arrow shows the distance the triangle has moved since the last collision test. Since we know that our triangle intersection routines will inflate the triangle by the radius of the sphere, we know the swept sphere can now be treated as a ray. This is the red line in Figure 12.44. Although the ray does not intersect the triangle in its previous position, it certainly does in its current position. We need to extend the ray such that it compensates for the movement of the triangle from its previous position to the current position. If we do this, we can simply test the ray against the triangle in its previous position. Once we have the intersection point, we can then shift it back by the distance the triangle has moved. This will give us the final intersection point (Modified Dest) along the ray such that the sphere is now resting against the polygon in its current position. More importantly, it will also catch the awkward case where the movement of the triangle was so large between frames, that it is completely bypassed by the ray. That is, we will develop a method that will always reliably catch the situation when the moving object intersects the sphere (even if the sphere is not moving).

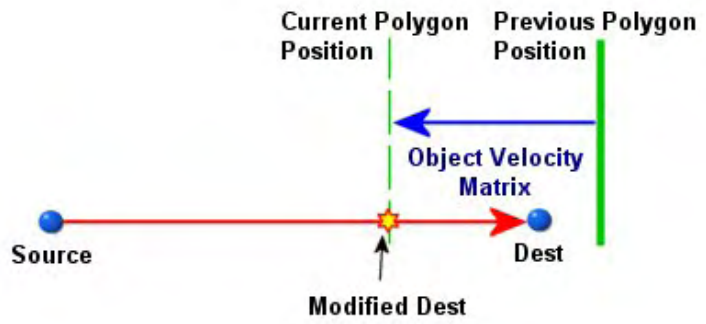


Figure 12.44

The process we are about to discuss will be used to alter the swept sphere's velocity vector for a given dynamic object. Once we have discussed these intricacies, we will then examine the complete modified EllipsoidIntersectScene function with dynamic object support.

As the EllipsoidIntersectScene function is working with eSpace variables, we have to temporarily undo this to calculate the amount of shift we have to apply to the velocity.

```
vecEndPoint = (Vec3VecScale(eCenter, Radius) + Vec3VecScale(eVelocity, Radius));
```

All we are doing here is calculating Dest in the above diagram in world space. By scaling the eSpace sphere center position and the eSpace velocity vector by the radius of the ellipsoid, we scale these back into world space. We then add the world space velocity vector that was input into the function to the world space sphere center position to get the position that our ellipsoid would like to move to in the world (if no collisions block its path). We store this desired destination position in the local variable vecEndPoint. This is the world space position that the application (or the previous response step) would like to move to the ellipsoid to.

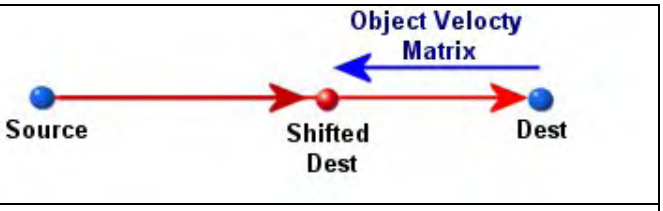


Figure 12.45

Our next step will be to take this world space destination position and transform it by the velocity matrix of the current dynamic object we are testing. In the case of our example shown in Figure 12.44, the velocity matrix describes the movement illustrated by the blue arrow. By transforming the destination position by this matrix we will shift the destination back along the ray. This shifted destination is illustrated in Figure 12.45. Notice how we are ultimately subtracting the movement of the object currently being tested from the end of our ray even though the position of the triangle in its previous location is past the end of the ray. This shifted destination position does not tell us anything quite yet, but bear with us. Below we show the code that shifts the ray end point back along the ray using the objects velocity matrix.

```
// Transform the end point
D3DXVec3TransformCoord( &eAdjust, &vecEndPoint, &pObject->VelocityMatrix );
```

At this point, Shifted Dest in Figure 12.45 is stored in the local variable eAdjust. We will now subtract the original end point of the ray from this shifted dest.

```
// Translate back so we have the difference
eAdjust -= vecEndPoint;
```

What we have a this point is a vector pointing in the opposite direction to the original velocity vector of our sphere whose length is equal to the distance moved by the dynamic object between its previous and current positions. We illustrate this in Figure 12.46. **Source** shows the original start point of the ray (the center of the ellipsoid) and **Shifted Dest** describes the position stored in eAdjust prior to the above line being executed. When we subtract the original ray end position from eAdjust we are subtracting the entire length of the original velocity vector from the shifted ray end point. This gives us an adjustment vector pointing in the opposite direction to the ray. This is shown in Figure 12.46 as the short blue arrow pointing from the ray origin (Source) to the left.

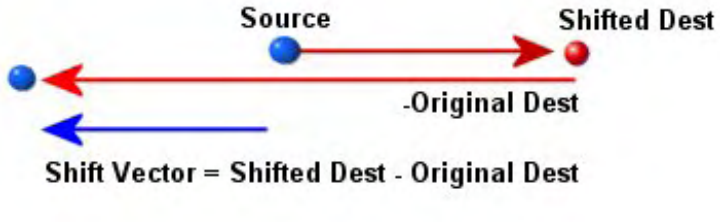


Figure 12.46

The next phase is the trick. We take this adjustment vector, transform it back into eSpace and then subtract it from the original velocity vector. Since the velocity vector and the adjustment vector are facing in opposite directions, subtracting the adjustment vector actually *adds it* to the original velocity vector. This extends the velocity vector of our swept sphere by the distance and direction the object has moved since the last collision test. Although we are illustrating just translational movement here this also works with rotations of the dynamic object.

```
// Scale back into ellipsoid space
eAdjust  = Vec3VecScale( eAdjust, InvRadius );

// Add to velocity
eVelocity-=eAdjust;
```

Figure 12.47 shows what our new velocity vector looks like. As we can see, it does indeed intersect the triangle in its previous position. We have altered the direction and length of velocity vector so that we can extrude it in the direction the object has moved since the last update and perform collision tests in its previous position.

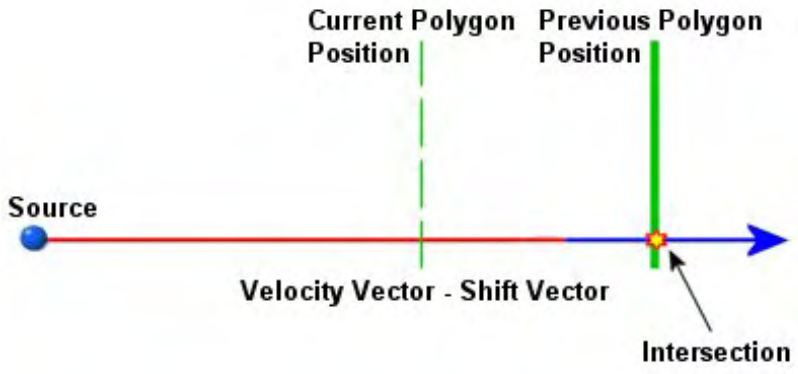


Figure 12.47

Now that we have the modified velocity vector, we can pass it into the SphereIntersectTriangle function for every triangle in the dynamic object. We do this for each dynamic object, searching for the intersection with the smallest interval. Once we have this new center position, what we actually have is the new center position of the unit sphere such that its surface is touching but not penetrating the closest intersecting triangle in the current dynamic object being tested. Of course, this is the position of the sphere resting against the triangle in its previous position and not its current position. All we have to do to correct this is take the resulting sphere center position and subtract from it the adjustment vector we initially added on. Remember, the adjustment vector is facing in the opposite direction from the velocity vector (in this example), so we actually add the adjustment vector to the resulting sphere center position to subtract it back.

```
// Translate back
Intersections[i].NewCenter      += eAdjust;
Intersections[i].IntersectPoint += eAdjust;
```

Above you can see that after we have added the new center position and the actual intersection point on the surface of the sphere to an intersection structure (to send back to the response step), we transform them back along the ray by the adjustment vector. Since this is the distance the triangle has moved from its previous position (the one we tested against), this shifts the sphere center position back so that it is now positioned on the shifted plane of the triangle in its current position instead of on the shifted plane of the triangle in its previous position. This gives us our final sphere center position (Modified Dest in Figure 12.47). As you can see, by calculating the adjustment vector, performing the intersection in the object's previous space, and then subtracting that adjustment vector from the intersection point, we essentially find out how much the sphere has to be moved back out of the way if a collision does occur.

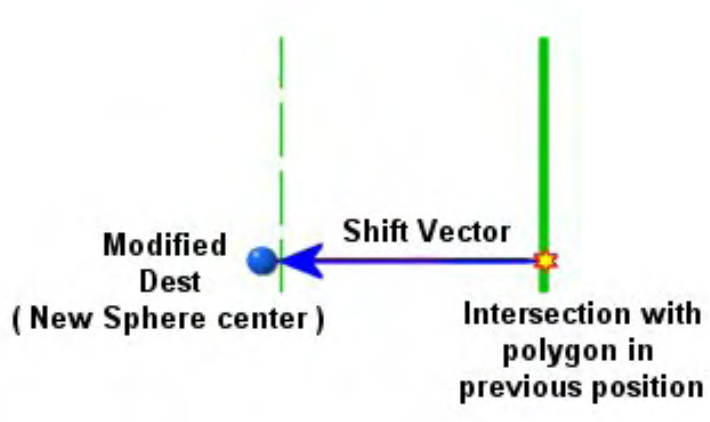


Figure 12.47

Once we have performed these same steps for every dynamic object and have found the closest intersecting triangle (or not), we then move on and test the original swept sphere against the static collision geometry using the methods explained earlier in the chapter. At the end of doing both the dynamic and static geometry tests, we will have a new ellipsoid position that we can return to the response step.

12.12.2 Integrating Dynamic Objects

In this final part of the textbook we will examine the code to the modified collision detection and response phases. The utility functions that add objects to the collision database and the way in which the application intersects with the collision system will all be discussed in the workbook. In this section, we will focus on updating the CollideEllipsoid and EllipsoidIntersectScene functions to handle both dynamic and static objects. Couple this knowledge with the fact that we have already discussed how the ObjectSetUpdated function works, and you should be able to understand everything in the workbook with very little trouble.

The first thing we will do is add a new generic function to our CCollision class that will remove the redundancy that would otherwise creep into the EllipsoidIntersectScene function now that we have many vertex and triangle buffers to process. The vertex and triangle data for each static triangle will all be stored in two STL vectors within the class namespace. These are the arrays that the EllipsoidIntersectScene function will need to extract geometry from when sweeping the sphere against

the static scene geometry. However, each dynamic object will also have its own buffers of triangle and vertex data and each of these will have to be tested too. Therefore, instead of repeating this code in the EllipsoidIntersectScene function, we will write a generic function called EllipsoidIntersectBuffers. This function can then be passed the buffer of information that needs to be processed by the EllipsoidIntersectScene function. For example, we will call this function for each dynamic object in turn, passing in a pointer to its vertex and triangle buffers. The function will loop through each triangle in the buffer and call our SphereIntersectTriangle function. It will return the new position of the sphere based on the closest collision interval. The function can be called once for each dynamic object, each time being passed the vertex and triangle buffers of the dynamic object for testing. All the while it will be keeping track of the closest intersection. Once this function has been called to process the buffer for each dynamic object, it can be called one final time and passed the vertex and triangle buffers containing all the static collision geometry.

The CCollision class has many helper functions to make registering objects with the collision database fast and easy. Below, we will look at just the member variables in the CCollision class so you get a feel for how it all works. Note that all the structures we have discussed so far are defined in the class.

```
class CCollision
{
public:

    //-----------------------------------------------------------------------
    // Public, class specific, Typedefs, structures & enumerators
    //-----------------------------------------------------------------------
    struct CollTriangle
    {
        ULONG       Indices[3];           // Indices referencing the triangle
                                          // vertices from our vertex buffer
        D3DXVECTOR3 Normal;               // The cached triangle normal.
        USHORT      SurfaceMaterial;      // The surface material index
                                          // assigned to this triangle.
    };

    // Vectors (STL Arrays) to contain triangle and vertex collision structures.
    typedef std::vector<CollTriangle>   CollTriVector;
    typedef std::vector<D3DXVECTOR3>    CollVertVector;

    struct DynamicObject
    {
        CollTriVector  *pCollTriangles;   // The triangle data referencing
                                          // the vertices below.
        CollVertVector *pCollVertices;    // The vertices defining
                                          // the level geometry.
        D3DXMATRIX     *pCurrentMatrix;   // A pointer to the actual matrix,
                                          // updated by external sources
        D3DXMATRIX      LastMatrix;       // The matrix recorded on the latest test
        D3DXMATRIX      VelocityMatrix;   // The difference for this frame
                                          // between our two matrices.
        D3DXMATRIX      CollisionMatrix;  // The space in which the collision
                                          // against this object will be performed.
        long            ObjectSetIndex;   // The index of the set of objects
                                          // this belongs to.
        bool            IsReference;      // Is this a reference object?
```

```
    };

    // Vector of a type to contain dynamic objects
    typedef std::vector<DynamicObject*> DynamicObjectVector;

    struct CollIntersect
    {
        D3DXVECTOR3     NewCenter;
        D3DXVECTOR3     IntersectPoint;
        D3DXVECTOR3     IntersectNormal;
        float           Interval;
        ULONG           TriangleIndex;  // The index of the triangle
                                        // we are intersecting
        DynamicObject * pObject;        // If we hit a dynamic object,
                                        // it will be stored here.
    };

private:
    CollTriVector       m_CollTriangles;        // The triangle data referencing
                                                // the vertices below.
    CollVertVector      m_CollVertices;         // The vertices defining
                                                // the level geometry.
    DynamicObjectVector m_DynamicObjects;       // The dynamic object structures.
    USHORT              m_nMaxIntersections;    // The total number of
                                                // intersections we should record
    USHORT              m_nMaxIterations;       // The maximum number of collision
                                                // test iterations we bail
    CollIntersect      *m_pIntersections;       // Internal buffer for storing
                                                // intersection information
    D3DXMATRIX          m_mtxWorldTransform;    // When adding primitives, this
                                                // transform will be applied.
    long                m_nLastObjectSet;       // The last object set index used.
};
```

As you can see, the member variables are declared at the bottom. Let us quickly discuss them.

CollTriVector m_CollTriangles
CollVertVector m_CollVertices

These two STL vectors store the static triangle and vertex data. Every object that is registered with the collision system that is not flagged as a dynamic object will have its triangle and vertex data added to these two vectors. The first of the two shown above is a vector containing CollTriangle structures and the second contains the D3DXVECTOR3 structures. Feeding these two vectors into our EllipsoidIntersectBuffers function (which we will write in a moment) will test the swept sphere against every triangle in the static scene geometry database.

DynamicObjectVector m_DynamicObjects

This is an STL vector that contains DynamicObject structures. Every object that is registered with the collision system as a dynamic object will be added to this vector. Each dynamic object structure contains its own triangle and vertex arrays (STL vectors). When a frame hierarchy (an actor) is registered with the collision system as a dynamic object, a dynamic object structure will be created and added to this vector for every mesh in the frame hierarchy. Each dynamic object added in this way will belong to the same object set and will be assigned the same object set index number.

USHORT m_nMaxIntersections
CollIntersect *m_pIntersections
These two variables control the size of an array of CollIntersect structures that are used temporarily by the detection phase to return information back to the response step about the closest intersecting triangles.

USHORT m_nMaxIterations
This member contains the maximum number of iterations of detection/response we wish to execute before we settle on the closest intersection and forcibly spend the remaining velocity. There are occasions when the ellipsoid could get stuck in a sliding loop chewing up CPU cycles as it bounces between surfaces. When this number of iterations is reached, we admit defeat and settle for pushing the ellipsoid to the closest intersect point.

D3DXMATRIX m_mtxWorldTransform
This matrix will describe a world transform to the collision system that should be used to transform one or more model space static meshes we are registering. The system will use this matrix as it is filling its vertex array with data from that model space mesh. This can be useful when loading referenced mesh from a file for example. You should be mindful to set this back to its default state of identity after you have finished using it since all future objects you register will also be transformed by this same matrix (i.e., it is a system level matrix).

long m_nLastObjectSet
This variable contains the number of the last object set index that was issued to a dynamic object. Every time a new object set is added, this value is incremented and assigned to the object as a means of identification. This does not necessarily describe the number of dynamic objects currently registered with the collision system. It can also apply to groups of dynamic objects that should be updated together.

As you can see, for such a complex system, it really does not have many variables to manage. Let us now see the code that has been modified to facilitate dynamic object integration. Luckily, our core intersection routines are exactly the same. The SphereIntersectTriangle function and all the sub-intersection routines it calls are unchanged. From their point of view, we are still just sweeping the sphere against a triangle, even when dealing with dynamic objects. The functions that have changed are the CollideEllipsoid and the EllipsoidIntersectScene functions. We have added an additional helper function which we will look a first. It is called EllipsoidIntersectBuffers.

12.13 The EllipsoidIntersectBuffers Function

The EllipsoidIntersectBuffers function will handle the intersection determination between a swept sphere and an arbitrary buffer of triangle data. Most of this code will be instantly recognizable as it existed in the previous EllipsoidIntersectScene function that we covered earlier. All we have done is extract the inner workings of that function into a separate function that can be used to sweep the sphere against triangles in the static scene database and the triangle buffers of each dynamic object. This

function will be called by the EllipsoidIntersectScene function, so let us first take a look at its parameter list. Most of these parameters will be familiar to us as they are simply the values that were passed into the EllipsoidIntersectScene function.

```
ULONG CCollision::EllipsoidIntersectBuffers( CollVertVector& VertBuffer,
                                             CollTriVector& TriBuffer,
                                             const D3DXVECTOR3& eCenter,
                                             const D3DXVECTOR3& Radius,
                                             const D3DXVECTOR3& InvRadius,
                                             const D3DXVECTOR3& eVelocity,
                                             float& eInterval,
                                             CollIntersect Intersections[],
                                             ULONG & IntersectionCount,
                                             D3DXMATRIX * pTransform /*= NULL*/ )
```

The first two parameters are new. They allow us to pass in the vertex and triangle buffers (STL vectors) we would like to sweep the sphere against. In our collision system, these parameters will be used either to pass the vertex and triangle buffers of the static scene geometry or the vertex and triangle buffers of a given dynamic object. The eCenter parameter is the current eSpace position of the ellipsoid that is about to be moved. The Radius and InvRadius vectors contain the radii and inverse radii of this ellipsoid. The InvRadius vector will be used to scale the vertex positions of each triangle into eSpace while the Radius vector will be used to scale the normal of each triangle into eSpace (remember the transform into eSpace works the opposite way for direction vectors). We also pass in the eSpace velocity vector describing the direction and distance we would like to move the sphere. We pass by reference the current *t* value of intersection. This will be passed to each of the intersection routines and collisions will only be considered if they have a smaller *t* value than what is currently stored. When the function returns, if a closer intersection was found than the one already stored in this variable, it will be overwritten with the new *t* value for the intersection.

As the eighth and ninth parameters we pass in the CollIntersect array which intersection results will be stored in and an output variable for a count of those items. As we have seen, this is the transport mechanism for returning the information back to the response phase. As discussed earlier, we must be mindful of the fact that this array may already hold intersection information. Recall that as soon as we find a closer intersection, we add this to the first element in the array and reset the counter to 1.

The final parameter is a pointer to a matrix that is used when processing the buffers of dynamic objects. The vertices in a dynamic object's vertex list are in model space, so before this function performs sphere/triangle tests on dynamic object geometry buffers, it must transform the triangle vertices into world space. When this function is called to perform collision testing against the static arrays, this parameter will be set to NULL. For dynamic object intersection testing, we will pass in the object's CollisionMatrix. As previously discussed, this is a world space transformation matrix that will transform the vertices of the dynamic object into their previous state.

Finally, this function will return an integer that describes the index into the intersection array where the intersection structures added by this function begin. For example, if this function was passed a CollIntersect array that contained five elements and another three were added, this function would return 5 to the caller and increment the intersection count to 8 (5 is the index of the first element in the array that was added by this function assuming zero-based offsets). If this function finds an intersection that is

smaller than the interval values of the intersections currently stored in the array, the function will return 0 since the array will be completely overwritten from the beginning with the new closer intersection information.

Let us now have a look at this function a few sections at a time.

The first thing we do is store the current intersection count that was passed in (FirstIndex). This contains the current end of the list and the index where any additional structures will be added by this function, assuming that a closer intersection time is not found. It is the value of FirstIntersect that will be returned from this function. It will be set to zero later in the function if we determine a closer intersection and start building the list from the beginning again.

```
{
    D3DXVECTOR3 ePoints[3], eNormal;
    D3DXVECTOR3 eIntersectNormal, eNewCenter;
    ULONG       Counter, NewIndex, FirstIndex;
    bool        AddToList;

    // FirstIndex tracks the first item to be added the intersection list.
    FirstIndex = IntersectionCount;

    // Iterate through our triangle database
    CollTriVector::iterator Iterator = TriBuffer.begin();
    for ( Counter = 0; Iterator != TriBuffer.end(); ++Iterator, ++Counter )
    {
        // Get the triangle descriptor
        CollTriangle * pTriangle = &(*Iterator);
        if ( !pTriangle ) continue;

        // Requires transformation?
        if ( pTransform )
        {
            // Get transformed points
            D3DXVec3TransformCoord(&ePoints[0], &VertBuffer[pTriangle->Indices[0]],
                                   pTransform );
            D3DXVec3TransformCoord(&ePoints[1], &VertBuffer[pTriangle->Indices[1]],
                                   pTransform );
            D3DXVec3TransformCoord(&ePoints[2], &VertBuffer[pTriangle->Indices[2]],
                                   pTransform );

            // Get points and transform into ellipsoid space
            ePoints[0] = Vec3VecScale( ePoints[0], InvRadius );
            ePoints[1] = Vec3VecScale( ePoints[1], InvRadius );
            ePoints[2] = Vec3VecScale( ePoints[2], InvRadius );

            // Transform normal and normalize
            // (Note how we do not use InvRadius for the normal)
            D3DXVec3TransformNormal( &eNormal, &pTriangle->Normal, pTransform );
            eNormal = Vec3VecScale( eNormal, Radius );
            D3DXVec3Normalize( &eNormal, &eNormal );

        } // End if requires transformation
```

Looking at the above code you can see that we create an iterator to access the elements of the STL vector and begin to loop through every triangle in the passed buffer. We then store a pointer to the current triangle in the local pTriangle pointer for ease of access. If for some reason this is not a valid triangle, we continue to the next iteration of the loop (a simple safety test).

We then test to see if the pTransform parameter is not NULL. If it is not, then it means that a matrix has been passed and the vertices of each triangle will need to be transformed by it prior to being intersection tested against the sphere. As you can see, we fetch the three vertices of the current triangle from the passed vertex buffer and transform them by this matrix. This code will only be executed if we are calling this function to process a dynamic object's geometry buffer. We store the transformed results of each world space vertex in the ePoints local array.

Next we scale each of the triangle vertices by the inverse radius of the ellipsoid, transforming these three vertices into eSpace. We then fetch the normal of the triangle and scale it from model space into world space using the passed transformation matrix (the CollisionMatrix of the dynamic object). Once we have the normal in world space, we scale it by the radius of the ellipsoid, transforming it into eSpace as well. We finally normalize the result.

If a valid transformation matrix was not passed, such as when processing the static geometry buffers, the triangles in those buffers are assumed to already be in world space and thus require only the transformation into eSpace.

```
        else
        {
            // Get points and transform into ellipsoid space
            ePoints[0] = Vec3VecScale( VertBuffer[ pTriangle->Indices[0] ],
                                       InvRadius );
            ePoints[1] = Vec3VecScale( VertBuffer[ pTriangle->Indices[1] ],
                                       InvRadius );
            ePoints[2] = Vec3VecScale( VertBuffer[ pTriangle->Indices[2] ],
                                       InvRadius );

            // Transform normal and normalize
            // (Note how we do not use InvRadius for the normal)
            eNormal = Vec3VecScale( pTriangle->Normal, Radius );
            D3DXVec3Normalize( &eNormal, &eNormal );

        } // End if no transformation
```

At this point we have the triangle in eSpace, so we use our SphereIntersectTriangle function to perform the intersection test. If the function returns true, an intersection was found and we are returned the collision normal in eSpace (eIntersectNormal) and the new *t* value (eInterval).

```
        // Test for intersection between unit sphere and ellipsoid space triangle
        if ( SphereIntersectTriangle( eCenter, 1.0f, eVelocity,
                                      ePoints[0], ePoints[1], ePoints[2],
                                      eNormal, eInterval, eIntersectNormal ) )
        {
            // Calculate our new sphere center at the point of intersection
            if ( eInterval > 0 )
```

```
            eNewCenter = eCenter + (eVelocity * eInterval);
        else
            eNewCenter = eCenter - (eIntersectNormal * eInterval);
```

If the function returns true we calculate the new sphere center in eSpace. If eInterval is larger than zero then it means the intersection happened along the ray and the position can be calculated by scaling the velocity vector by the interval and adding the resulting vector to the original sphere center. If the collision interval is negative, then the sphere was embedded in the triangle and needs to be moved backwards such that this is no longer the case. Under these circumstances, the actual (negative) distance to the shifted plane is stored in eInterval and we can move the sphere center position back on to the shifted plane by moving along the plane normal by that distance.

In the next section of code we test to see if the interval is smaller than the intervals of the collision structures currently stored in the array. If so, it means we have found a triangle that intersects the ray closer to the origin than any currently stored in the array. When this is the case we set NewIndex to zero (which contains the index of the element in the array where the information about this intersection will be stored). As we have just invalidated all the information currently stored in the array, we store this in element zero and reset the intersection count back to 1. We also set FirstIndex to zero. Remember, this will be returned from the function as the index of the first structure added by this function to this array.

```
        // Where in the array should it go?
        AddToList = false;
        if ( IntersectionCount == 0 || eInterval < Intersections[0].Interval )
        {
            // We either have nothing in the array yet,
            // or the new intersection is closer to us
            AddToList         = true;
            NewIndex          = 0;
            IntersectionCount = 1;

            // Reset, we've cleared the list
            FirstIndex        = 0;

        } // End if overwrite existing intersections
```

If the interval is equal (with tolerance) to the intervals of the structures already stored in the array, we will add this new intersection information to the back of the array.

```
        else if ( fabsf( eInterval - Intersections[0].Interval ) < 1e-5f )
        {
            // It has the same interval as those in our list already, append to
            // the end unless we've already reached our limit
            if ( IntersectionCount < m_nMaxIntersections )
            {
                AddToList         = true;
                NewIndex          = IntersectionCount;
                IntersectionCount++;

            } // End if we have room to store more

        } // End if the same interval
```

If the collision interval for the current triangle was either smaller or the same as the intervals currently stored, the AddToList boolean would have been set to true; otherwise, it will still be false, nothing will happen, and the following code block will be skipped. This is the code block that adds the collision information to the array (explained earlier in the EllipsoidIntersectScene discussion).

```
            // Add to the list?
            if ( AddToList )
            {
                Intersections[ NewIndex ].Interval = eInterval;
                Intersections[ NewIndex ].NewCenter = eNewCenter +
                                                      (eIntersectNormal * 1e-3f);
                Intersections[ NewIndex ].IntersectPoint = eNewCenter –
                                                           eIntersectNormal;
                Intersections[ NewIndex ].IntersectNormal = eIntersectNormal;
                Intersections[ NewIndex ].TriangleIndex = Counter;
                Intersections[ NewIndex ].pObject = NULL;

            } // End if we are inserting in our list

        } // End if collided

    } // Next Triangle

    // Return hit.
    return FirstIndex;
}
```

12.14 The Revised EllipsoidIntersectScene Function

We will now examine the code to the modified EllipsoidIntersectScene function that adds support for dynamic objects and uses the previous function to perform the intersection with the various geometry buffers. This function is essentially the detection phase of our collision system. It is called by CollideEllipsoid for each movement of the ellipsoid along the initial or slide vectors. Our CollideEllipsoid function first converts the sphere center and velocity into eSpace before passing them in.

While the detection function will stay pretty constant as its job is fairly generic, the CollideEllipsoid function is very application specific. You may wish to add different response handling functions to the CCollision class to make the ellipsoid act in a different way. Further, you may not even be using ellipsoids at all; instead you might be using a sphere of arbitrary size to bound your objects. In such a case, you would not want to needlessly perform the eSpace transformation code. Additionally you might not wish to use the CCollision system's response phase at all, but instead allow your application to make direct calls to EllipsoidIntersectScene and handle you own responses. When this is the case, you would not want to have to concern your application with feeding in eSpace values for the ellipsoid center and velocity. Preferably, you could use the EllipsoidIntersectScene function as a black box that does this for you. The same is true for output. You are not likely to want to get back eSpace values for the

intersection point, integration velocity, and the new sphere position. Ideally you would to feed EllipsoidIntersectScene world space values and get back world space values. So to make the function a little more generic, we will add two Boolean parameters that express whether the function is being passed eSpace input values and whether it should return eSpace values or perform the transformation back into world space prior to returning the information. Here is the new parameter list:

```
bool CCollision::EllipsoidIntersectScene(const D3DXVECTOR3& Center,
                                         const D3DXVECTOR3& Radius,
                                         const D3DXVECTOR3& Velocity,
                                         CollIntersect Intersections[],
                                         ULONG& IntersectionCount,
                                         bool bInputEllipsoidSpace /* = false */,
                                         bool bReturnEllipsoidSpace /* = false */ )
```

The final two parameters default to false, where the function will assume that it is being passed world space values and perform the transformation into eSpace using the passed radius vector. Because the final parameter is also set to false, this means the function will also convert the intersection information (stored in the passed CollIntersect array) back into world space prior to the function returning. While this makes the function a nice black box for third party calls, it should be noted that our CollideEllipsoid function (the function that calls this one) will transform the input values into eSpace and also requires eSpace values to be returned exactly as it did before. Therefore, when our CollideEllipsoid function calls this function, it will pass true for the final two parameters (the Center and Velocity vectors passed in will already be in eSpace).

The fourth and fifth parameters are the array and count of the CCollision object's CollIntersect array. It may seem strange to pass this array as a parameter to a CCollision function when it is already a class variable that the function would automatically have access to. We do this so that the caller can get back the results in their own array, should they wish not to use the complete collision detection and response system, but rather just the EllipsoidIntersectScene function.

Let us now talk about the new version of this function. The first thing we do is create the inverse radius vector that will be used to scale the passed center position and velocity into eSpace. If the bInputEllipsoidSpace parameter is set to false, the input variables are in world space and need to be transformed before we can continue. We copy the eSpace results into the local variables eCenter and eVelocity. If the bInputEllipsoidSpace parameter is set to true, the input variables are already in ellipsoid space and we simply copy them into the eCenter and eVelocity variables.

```
{
    D3DXVECTOR3 eCenter, eVelocity, eAdjust, vecEndPoint, InvRadius;
    float       eInterval;
    ULONG       i;

    // Calculate the reciprocal radius to prevent the many
    // divides we would need otherwise
    InvRadius = D3DXVECTOR3( 1.0f / Radius.x, 1.0f / Radius.y, 1.0f / Radius.z );

    // Convert the values specified into ellipsoid space if required
    if ( !bInputEllipsoidSpace )
    {
        eCenter   = Vec3VecScale( Center, InvRadius );
```

```
        eVelocity = Vec3VecScale( Velocity, InvRadius );
    } // End if the input values were not in ellipsoid space
    else
    {
        eCenter   = Center;
        eVelocity = Velocity;

    } // End if the input values are already in ellipsoid space
```

At this point we have our sphere center position and the velocity vector in eSpace. Next we set the eInterval variable to its maximum intersection interval of 1.0. This local variable will be fed into the EllipsoidIntersectBuffers function and will always contain the *t* value of the closest intersection we have found so far. When a closer intersection is found, the EllipsoidIntersectBuffers function will overwrite it. The initial value of 1.0 basically describes a *t* value at the end of the ray. If this is not altered by any of the collision tests, this means the velocity vector is free from obstruction. We also set the initial intersection count to zero as we have not identified any collisions yet.

```
    // Reset ellipsoid space interval to maximum
    eInterval = 1.0f;

    // Reset initial intersection count to 0 to save the caller having to do this.
    IntersectionCount = 0;
```

Now we will loop through each dynamic object in the m_DynamicObjects vector. For each one we will use its velocity vector to create the shift vector that is used to extend the length and direction of the velocity vector. This is the exact code we saw earlier when first discussing dynamic intersections. This process is also illustrated in Figures 12.43 through 12.47.

```
    // Iterate through our triangle database
    DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
    for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
    {
        DynamicObject * pObject = *ObjIterator;

        // Calculate our adjustment vector in world space.
        // This is our velocity adjustment for objects
        // so we have to work in the original world space.
        vecEndPoint = ( Vec3VecScale( eCenter, Radius ) +
                        Vec3VecScale( eVelocity, Radius ));

        // Transform the end point
        D3DXVec3TransformCoord( &eAdjust, &vecEndPoint, &pObject->VelocityMatrix );

        // Translate back so we have the difference
        eAdjust -= vecEndPoint;

        // Scale back into ellipsoid space
        eAdjust  = Vec3VecScale( eAdjust, InvRadius );

        // Perform the ellipsoid intersect test against this object
        ULONG StartIntersection = EllipsoidIntersectBuffers(
                                                *pObject->pCollVertices,
                                                *pObject->pCollTriangles,
```

```
                                                    eCenter,
                                                    Radius,
                                                    InvRadius,
                                                    eVelocity - eAdjust,
                                                    eInterval,
                                                    Intersections,
                                                    IntersectionCount,
                                                    &pObject->CollisionMatrix );
```

Notice how once we have calculated the shift vector (eAdjust), we subtract it from the end of the velocity vector to extend the ray in the opposite direction of the dynamic object's movement. We do this while passing the velocity vector into the EllipsoidIntersectBuffers function as the sixth parameter. Note also that for the first two parameters we pass in the vertex and triangle arrays of the current dynamic object being processed for collision. The function will return the index of the first intersection structure that the call added to the CollIntersect array. Remember, this function will be called for each dynamic object and therefore, many different calls to the function may result in intersection information being cumulatively added to the CollIntersect array. If the function found no additional intersections, StartIntersect will be equal to IntersectionCount on function return.

Now it is time to loop through any structures that were added by the above function call and translate them back by the adjustment vector. By doing this little trick, we are sure to catch the sphere if it is between the previous and current position of the moving triangle. We can then translate the sphere back by the movement of the triangle so it is pushed out of the way by the dynamic object.

```
        // Loop through the intersections returned
        for ( i = StartIntersection; i < IntersectionCount; ++i )
        {
            // Move us to the correct point (including the objects velocity)
            // if we were not embedded.
            if ( Intersections[i].Interval > 0 )
            {
                // Translate back
                Intersections[i].NewCenter      += eAdjust;
                Intersections[i].IntersectPoint += eAdjust;

            } // End if not embedded

            // Store object
            Intersections[i].pObject = pObject;

        } // Next Intersection

    } // Next Dynamic Object
```

Notice that at the bottom of the dynamic object loop, if an intersection has occurred with the object's buffers, we also store a pointer to the dynamic object in each CollIntersect structure. This might be useful if the calling function would like to know more information about which object was actually hit.

With all the dynamic objects tested for intersection and the closest intersection recorded in eInterval, we now have to test the static scene geometry for intersection with the swept sphere. We do this by simply calling the EllipsoidIntersectBuffers function one more time. This time we pass in the vertex and

triangle buffers containing the static scene geometry along with the CCollision object's m_CollVertices and m_CollTriangles members.

```
    // Perform the ellipsoid intersect test against our static scene
    EllipsoidIntersectBuffers(
                         m_CollVertices,
                         m_CollTriangles,
                         eCenter,
                         Radius,
                         InvRadius,
                         eVelocity,
                         eInterval,
                         Intersections,
                         IntersectionCount );
```

At this point, if any intersections have occurred with any geometry, the CollIntersect array will contain information about the closest intersection. The array may contain multiple entries if collisions occurred with multiple triangles at the exact same time. Either way, the first structure in this array will contain the new center position of the sphere. It is automatically returned to the response phase in CollideEllipsoid.

Before we return there is one final step that we will take. If bReturnEllipsoidSpace is set to false, then the function should transform all the eSpace values stored in the CollIntersect array into world space before returning. If this is the case, we loop through each intersection structure and apply the ellipsoid radius scale to make that so. Notice once again how we transform a positional vector from eSpace to world space by scaling it by the radii of the ellipsoid, but we transform a direction vector from eSpace to world space by scaling it by the inverse radii of the ellipsoid.

```
    // If we were requested to return the values in normal space
    // then we must take the values back out of ellipsoid space here
    if ( !bReturnEllipsoidSpace )
    {
        // For each intersection found
        for ( i = 0; i < IntersectionCount; ++i )
        {
            // Transform the new center position and intersection point
            Intersections[ i ].NewCenter =
                             Vec3VecScale( Intersections[ i ].NewCenter, Radius );
            Intersections[ i ].IntersectPoint =
                             Vec3VecScale( Intersections[ i ].IntersectPoint,
                                           Radius );

            // Transform the normal
            // (again we do this in the opposite way to a coordinate)
            D3DXVECTOR3 Normal =
                     Vec3VecScale( Intersections[ i ].IntersectNormal, InvRadius );
            D3DXVec3Normalize( &Normal, &Normal );

            // Store the transformed normal
            Intersections[ i ].IntersectNormal = Normal;

        } // Next Intersection

    } // End if !bReturnEllipsoidSpace
```

```
    // Return hit.
    return (IntersectionCount > 0);
}
```

The function simply returns true if the number of intersections found is greater than zero.

12.15 The Revised CollideEllipsoid Function

The CollideEllipsoid has also received a minor upgrade due to the integration of dynamic objects. The only major change involves the calculation of the integration vector that is returned back to the application.

As discussed earlier, the integration velocity vector parameter is an output parameter that will contain the new direction and speed of the object on function return. If no collisions have occurred, this will be a straight copy of the velocity vector passed in. However, while the input velocity vector is continually split at intersection planes and recalculated by projecting any remaining velocity past the point of intersection onto the slide plane, we do not wish the integration velocity to be diminished in this way. We should retain as much of its length as possible so that we can more accurately communicate the new direction and speed of the moving entity back to the application on function return.

Now we will have to take collisions with dynamic objects into account when calculating this vector. When the slide plane is on a dynamic object, we must once again calculate the adjustment vector in order to extend the integration velocity vector so that it can be projected onto the slide plane. However, remember that the intersection was performed between our velocity vector and the triangle of the dynamic object by adjusting the length of the ray to compensate for object movement. Since we now wish to project our integration velocity onto this same plane, we must also extend that vector in the same manner. Recall that we returned the intersection normal based on the dynamic object's previous position instead of its current one. The velocity matrix that describes the transition from the dynamic object's previous position to its current one may contain rotation as well. In short, we need to know that when we project the integration velocity vector onto the slide plane that the velocity vector is in the same *space* as the slide plane; otherwise the projection of the velocity vector onto this slide plane will be incorrect. This probably all sounds very complicated, but when you see the code you will realize that this is the exact same code we used to extend the velocity vector when testing for intersection against dynamic objects.

We have also added another feature to our CollideEllipsoid function. As we know, the CollIntersect array that is returned from the EllipsoidIntersectScene function contains the information about the closest collisions. If this array contains more than one element, then it means the ellipsoid intersected multiple geometry constructs at the same time. We also know that each CollIntersect structure contains within it, a vector describing the point of intersection on the surface of the ellipsoid. It is sometimes handy to have two vectors returned that describe the maximum and minimum collision extents on the surface of the ellipsoid. For example, our application will know that if the player is walking on the ground, then we should always have at least of one collision (the collision between the bottom of the ellipsoid and the floor polygon). We know that if the minimum y component of all the intersection

points that occurred on the surface of the ellipsoid is very near to -Radius.y, the height of this point places it at the ground level with respect to the ellipsoid. Our application uses this to set a state that indicates that the player is on the ground and friction should be applied to diminish the velocity of the player.

This state is also tested during user input when the user presses the CTRL key to see whether the player should be allowed to jump -- a player that is in mid-air and currently falling should not be able to jump. Remember that an intersection point with a y component of zero would describe a point that is aligned with the center of the ellipsoid along the y axis. The top and bottom of the ellipsoid is described by +Radius.y and –Radius.y from the center of the ellipsoid, respectively.

Similarly, we know that if the y component of the lowest intersection point on the surface of the ellipsoid is zero, then the intersection point is aligned vertically with the center of the ellipsoid. This would mean that the player is currently in the air, and if gravity is being applied, the player will be falling. The collision point would also suggest to the application that an object has collided into a wall as the player is falling.

Finally, we know that when we have an intersection point nearing a y value of zero, the intersection point is aligned with the center of the sphere along the y axis. If you think about this for a moment, you will see that this means the player is hitting something almost head on, and its velocity should be set to zero. In fact, as we walk our ellipsoid into steeper inclines, we will find intersection points that are near the center of the ellipsoid vertically. Our application will use this information to slow the velocity of the player so that the steeper the hill, the slower the speed of the player as it ascends.

How these minimum and maximum collision extents are used by the application will be discussed in the workbook since it is not part of the collision detection system proper. However, the CollideEllipsoid function will record the minimum and maximum extents of the intersection points and return them via two output vectors that are passed as parameters.

Let us have a look at the new parameter list to this function. The only difference is that we have added these two new parameters to the end. These two vectors will be filled by the function with the minimum and maximum x, y, and z components of all intersection points. Thus these two vectors describe a bounding box within which all the intersection points are contained.

```
bool CCollision::CollideEllipsoid( const D3DXVECTOR3& Center,
                                   const D3DXVECTOR3& Radius,
                                   const D3DXVECTOR3& Velocity,
                                   D3DXVECTOR3& NewCenter,
                                   D3DXVECTOR3& NewIntegrationVelocity,
                                   D3DXVECTOR3& CollExtentsMin,
                                   D3DXVECTOR3& CollExtentsMax )
{
```

This is the function that is called by the application to calculate a movement update for an entity. The first parameter is the world space position of the center of the ellipsoid and the second is its radius vector. The Velocity vector describes the direction and distance the application would like to move the ellipsoid. On function return, NewCenter will describe the new world space position of the ellipsoid that

is free from obstruction. It can be used by the application to update the position of the entity which the ellipsoid is approximating. On function return, the NewIntegrationVelocity parameter will contain the new direction and speed which the object should be heading after collisions and their responses have been taken into account. The final two parameters will contain two vectors describing the bounding box of intersection points.

We will look at the function a bit at a time, although most of this code will be familiar from the earlier version. First we add the passed velocity to the passed sphere center point to describe the world space position we would like to move the sphere to. We store the resulting position in the local variable vecOutputPos. We also take the passed velocity and store it in the local variable vecOutputVelocity. It is this output velocity that will contain the integration velocity we wish to return to the application. At the moment we set it to the current velocity because if no intersections occur, the velocity should not be changed.

```
    D3DXVECTOR3 vecOutputPos, vecOutputVelocity, InvRadius, vecNormal, vecExtents;
    D3DXVECTOR3 eVelocity, eInputVelocity, eFrom, eTo;
    D3DXVECTOR3 vecNewCenter, vecIntersectPoint;
    D3DXVECTOR3 vecFrom, vecEndPoint, vecAdjust;
    ULONG       IntersectionCount, i;
    float       fDistance, fDot;
    bool        bHit = false;

    vecOutputPos        = Center + Velocity;
    vecOutputVelocity   = Velocity;
```

Next we initialize the minimum and maximum extents vectors to the initial extremes. That is, we set the minimum vector to the maximum extents of the ellipsoid and set the maximum vector to the minimum extents of the ellipsoid. This will make sure that minimum extent starts as large as possible and will be set to the first intersection point we initially find, and vice versa. We have seen code like this when initializing bounding boxes in the past.

```
    // Default our intersection extents
    CollExtentsMin.x  = Radius.x;
    CollExtentsMin.y  = Radius.y;
    CollExtentsMin.z  = Radius.z;
    CollExtentsMax.x  = -Radius.x;
    CollExtentsMax.y  = -Radius.y;
    CollExtentsMax.z  = -Radius.z;
```

Now we create the inverse radius vector and scale the sphere center position and the velocity vector into eSpace. We store the current eSpace sphere position in eFrom, the eSpace velocity in eVelocity, and the eSpace desired destination position in eTo.

```
    // Store ellipsoid transformation values
    InvRadius = D3DXVECTOR3( 1.0f / Radius.x, 1.0f / Radius.y, 1.0f / Radius.z );

    // Calculate values in ellipsoid space
    eVelocity = Vec3VecScale( Velocity, InvRadius );
    eFrom     = Vec3VecScale( Center, InvRadius );
    eTo       = eFrom + eVelocity;
```

```
    // Store the input velocity for jump testing
    eInputVelocity = eVelocity;
```

Notice that we copy the eSpace velocity passed into the function into the eInputVelocity local variable. As discussed earlier, we may need this later if, after the maximum number of collision iterations, we still cannot resolve a final position. When this is the case, we will just use the initial velocity vector passed in to run the detection phase one last time to get a position which we will settle for (no sliding).

We now loop through the maximum number of iterations we wish to spend finding the correct position. Inside the loop, we see that if the current length of our velocity vector is zero (with tolerance) we have no more sliding left to do and can exit. Otherwise, we call the EllipsoidIntersectScene function to test for collisions along the current velocity vector. Notice how we pass true for the two boolean parameters, indicating that we will be passing in eSpace values and would like to have eSpace values returned.

```
    // Keep testing until we hit our max iteration limit
    for ( i = 0; i < m_nMaxIterations; ++i )
    {
        // Break out if our velocity is too small
        // (optional but sometimes beneficial)
        //if ( D3DXVec3Length( &eVelocity ) < 1e-5f ) break;

        if ( EllipsoidIntersectScene( eFrom, Radius, eVelocity,
                                m_pIntersections, IntersectionCount, true, true ) )
        {
            // Retrieve the first collision intersections
            CollIntersect & FirstIntersect = m_pIntersections[0];

            // Calculate the WORLD space sliding normal
            D3DXVec3Normalize( &vecNormal,
                               &Vec3VecScale( FirstIntersect.IntersectNormal,
                                              InvRadius )
                             );
```

If the function returns true then the sphere collided with something along the velocity vector and we should access the new position of the sphere from the first CollIntersect structure in the array. We also transform the returned eSpace collision normal into world space and normalize the result. We will need to do some work with this when projecting the integration vector.

In the next section of code we would usually just project the current integration velocity vector onto the slide plane, but if this collision has occurred with a dynamic object, then we have to calculate the force and direction at which the dynamic object struck the ellipsoid so that we can add that to the current integration velocity. As it happens, we already know how to do this. When we performed intersection tests between the sphere and triangles of a dynamic object, you will recall how we create an adjustment vector that allows us to extend the ray and collide with the previous position of the object. This adjustment vector described the direction the object has moved since the previous update. If we calculate this adjustment vector in the same way, we have a vector that describes the speed and distance of movement the dynamic object has moved as shown below.

```
            // Did we collide with a dynamic object
            if ( FirstIntersect.pObject )
```

```
            {
                DynamicObject *pObject = FirstIntersect.pObject;

                // Calculate our adjustment vector in world space. This is for our
                // velocity adjustment for objects so we have to work in the
                // original world space.
                vecFrom     = Vec3VecScale( eFrom, Radius );
                vecEndPoint = vecFrom + vecOutputVelocity;

                // Transform the end point
                D3DXVec3TransformCoord( &vecAdjust, &vecEndPoint,
                                        &pObject->VelocityMatrix );

                // Translate back so we only get the distance.
                 vecAdjust -= vecEndPoint;
```

The code above is not new to us -- we created a shift vector (eAdjust) in exactly the same way when performing the intersection tests. The adjustment vector describes the movement of the dynamic object from its previous state to its current state; the period in which the collision with the sphere happened. This vector tells us the speed which the dynamic object was traveling from its previous position to its new position and we can use this to approximate an applied force on our sphere as a result of the impact. After we have projected the current integration velocity vector onto the slide plane, we have a new integration vector facing in the slide direction:

```
                // Flatten out to slide velocity
                fDistance = D3DXVec3Dot( &vecOutputVelocity, &vecNormal );
                vecOutputVelocity -= vecNormal * fDistance;
```

We will then calculate the angle between the adjustment vector and the slide plane normal. Since the slide plane normal describes the orientation of the point of impact on the triangle, and the adjustment vector describes the speed and direction of the movement of the triangle when it hit the sphere, performing the dot product between them will give us with a distance value that gets larger the more head-on the collision between the two objects. We will use this distance value to add a force to the integration velocity that pushes the object away from the triangle in the direction of the triangle normal.

```
                // Apply the object's momentum to our velocity along the
                // intersection normal
                fDot = D3DXVec3Dot( &vecAdjust, &vecNormal );
                vecOutputVelocity += vecNormal * fDot;

            } // End if colliding with dynamic object
```

Remember, even before we calculate the integration vector, the sphere has been correctly repositioned so that it has moved back with the colliding triangle and thus is not intersecting. By adding this force to the integration velocity that is returned to the application, it means that our application will continue to move the sphere back away from the point of impact for the next several frames. This is certainly not state of the art physics here, but it certainly provides a convincing enough effect to be suitable for our purposes in this demo.

The next code block is executed if the intersection information does not relate to a dynamic object. In this case we just project the current integration velocity onto the side plane, just as we did in the previous version of the function.

```
            else
            {
                // Generate slide velocity
                fDistance = D3DXVec3Dot( &vecOutputVelocity, &vecNormal );
                vecOutputVelocity -= vecNormal * fDistance;

            } // End if colliding with static scene
```

We have now modified the integration velocity vector and accounted for collisions with a static or scene triangle. The application can now be informed of the new direction the object should be heading when the function returns.

In the next section of code we grab the new eSpace position of the sphere and store it in the eFrom local variable. This is the new position of the sphere as suggested by the collision detection phase. We then calculate the slide vector that will become the velocity vector passed into the EllipsoidIntersectScene function in the next iteration of the collision loop. We subtract the actual position we were allowed to move to from the previous desired destination position which provides us with the remaining velocity vector after the point of intersection. This is then dotted with the plane normal, giving us the distance from the original desired destination position to the slide plane. We then subtract this distance from the original intended destination along the direction of the plane normal, thus flattening the remaining velocity onto the slide plane. This is stored in the eTo local variable which will be used in a moment to calculate the new slide velocity that will be one of the inputs for the next collision detection loop.

```
            // Set the sphere position to the collision position
            // for the next iteration of testing
            eFrom = FirstIntersect.NewCenter;

            // Project the end of the velocity vector onto the collision plane
            // (at the sphere centre)
            fDistance = D3DXVec3Dot( &( eTo - FirstIntersect.NewCenter ),
                                     &FirstIntersect.IntersectNormal );
            eTo -= FirstIntersect.IntersectNormal * fDistance;
```

We will now get the new sphere position and the intersection point on the surface of the sphere and transform them into world space. This will be done for the purpose of testing the intersection point against the minimum and maximum extents currently recorded. We adjust the relevant extents vector if the intersection point is found to outside the bounding box (i.e., we will grow the box to fit). To calculate the new world space sphere center position, we just transform the new eSpace sphere position returned from the collision detection function using the ellipsoid radius vector. To calculate the intersection point on the surface of the ellipsoid we simply scale the slide plane normal into world space by the radius vector and subtract the world space sphere position from it.

```
            // Transform the sphere position back into world space,
            // and recalculate the intersect point
            vecNewCenter       = Vec3VecScale( FirstIntersect.NewCenter, Radius );
```

```
            vecIntersectPoint  = vecNewCenter - Vec3VecScale( vecNormal, Radius );
```

If the calculation of the intersection point is bothering you, remember that in eSpace the sphere has a radius of one. Since a unit vector also has a length of one we know that in eSpace, the intersection point on the sphere can be found simply by subtracting the slide plane normal from the sphere center position. This will always generate a vector at a distance of 1.0 unit from the sphere center in the direction of the intersection point. Because we are transforming the unit normal by the radius vector, we also scale its length as well. The resulting vector will have a length equal to the world space distance from the center of the ellipsoid to the point on the surface of the ellipsoid. If we subtract this from the world space ellipsoid position, we have the intersection point in world space on the surface of the ellipsoid.

In the next section of code we subtract the ellipsoid center position to leave us with an intersection point that is relative to its center. We can now test the x, y, and z components of the intersection point against the corresponding components in the extents vectors and grow those vectors if we find a component of the intersection point vector that is outside the box. The result is a bounding box where there ellipsoid center is assumed to be at its center.

```
            // Calculate the min/max collision extents around the ellipsoid center
            vecExtents = vecIntersectPoint - vecNewCenter;
            if ( vecExtents.x > CollExtentsMax.x ) CollExtentsMax.x = vecExtents.x;
            if ( vecExtents.y > CollExtentsMax.y ) CollExtentsMax.y = vecExtents.y;
            if ( vecExtents.z > CollExtentsMax.z ) CollExtentsMax.z = vecExtents.z;
            if ( vecExtents.x < CollExtentsMin.x ) CollExtentsMin.x = vecExtents.x;
            if ( vecExtents.y < CollExtentsMin.y ) CollExtentsMin.y = vecExtents.y;
            if ( vecExtents.z < CollExtentsMin.z ) CollExtentsMin.z = vecExtents.z;
```

Next we create the slide vector that will become the velocity vector for the next iteration of the collision detection phase. In the eTo vector, we currently have the original desired destination point projected onto the slide plane. This describes the location we want to travel to in the next iteration. In eFrom we also have the new non-interpenetrating position of the ellipsoid returned from the previous call to EllipsoidIntersectScene. Therefore, by subtracting eFrom from eTo, we have our new velocity vector.

```
            // Update the velocity value
            eVelocity = eTo - eFrom;

            // We hit something
            bHit = true;

            // Filter 'Impulse' jumps (described in earlier version of function)
            if ( D3DXVec3Dot( &eVelocity, &eInputVelocity ) < 0 )
            { eTo = eFrom; break; }

        } // End if we got some intersections
        else
        {
            // We found no collisions, so break out of the loop
            break;

        } // End if no collision

    } // Next Iteration
```

Above we see the final section of code in the inner loop. Hopefully, we will have found a collision and broken from the above loop before the maximum number of iterations. Notice how we use the additional dot product to filter out back and forth vibration. Now you see why we made that additional copy of the initial velocity vector that was input into the function. We would need it later to test if we ever generate a slide vector in the response phase that is bouncing the ellipsoid back in the opposite direction than was initially intended. If we find this is the case, we set eTo equal to eFrom, making the slide vector zero length so that no more sliding is performed.

The next section of code is executed if an intersection occurred. If at any point in the detection loop above EllipsoidIntersectScene returned true, the local boolean variable bHit will have been set to true. Inside this conditional is another one that tests whether the number of loop iterations we executed is equal to the maximum number of iterations. If not, then it means we successfully resolved the response to our satisfaction and simply have to scale the new position of the sphere into world space.

```
    // Did we register any intersection at all?
    if ( bHit )
    {
        // Did we finish neatly or not?
        if ( i < m_nMaxIterations )
        {
            // Return our final position in world space
            vecOutputPos = Vec3VecScale( eTo, Radius );

        } // End if in clear space
```

However, if the loop variable *i* is equal to the maximum number of iterations allowed, it means we could not find a satisfactory position to slide to in the given number of iterations. When this is the case we simply call EllipsoidIntersectScene again with the original input values to get the closest non-intersecting position and store this in vecOutputPos after transforming it into world space. As you can see, if this section of code is executed, we have encountered the undesirable situation where the response step was repeatedly bouncing the sphere between two triangles. When this is the case, we forget about the slide step and just move the sphere along the velocity vector as much as possible and discard any remaining velocity.

```
        else
        {
            // Just find the closest intersection
            eFrom = Vec3VecScale( Center, InvRadius );

            // Attempt to intersect the scene
            IntersectionCount = 0;
            if ( EllipsoidIntersectScene( eFrom, Radius, eInputVelocity,
                                          m_pIntersections, IntersectionCount,
                                          true, true ) )
            {
                // Retrieve the intersection point in clear space,
                // but ensure that we undo the epsilon
                // shift we apply during the above call.
                // This ensures that when we are stuck between two
                // planes, we don't slowly push our way through.
                vecOutputPos = m_pIntersections[0].NewCenter –
```

```
                                    (m_pIntersections[0].IntersectNormal * 1e-3f);

                // Scale back into world space
                vecOutputPos = Vec3VecScale( vecOutputPos, Radius );

            } // End if collision
            else
            {
                // Don't move at all, stay where we were
                vecOutputPos = Center;
            } // End if no collision

        } // End if bad situation

    } // End if intersection found

    // Store the resulting output values
    NewCenter             = vecOutputPos;
    NewIntegrationVelocity = vecOutputVelocity;

    // Return hit code
    return bHit;
}
```

The final few lines of code simply store the new world space ellipsoid position in the NewCenter variable and store the integration velocity in the NewIntegrationVelocity vector. Since both of these vectors were passed by reference, the application will be able to access these values when the function returns. We conclude by returning the value of bHit, which will be set to either true or false depending on whether EllipsoidIntersectScene call ever returned true.

Conclusion

This has probably been a very challenging lesson for many of you as it was certainly one of our most complicated and most mathematically intense systems to date in the series. Do not be concerned if you find that you need to re-read parts of this chapter again in order to fully grasp what is happening. It is recommended that you take your time with the material and try to keep the actual source code in hand.

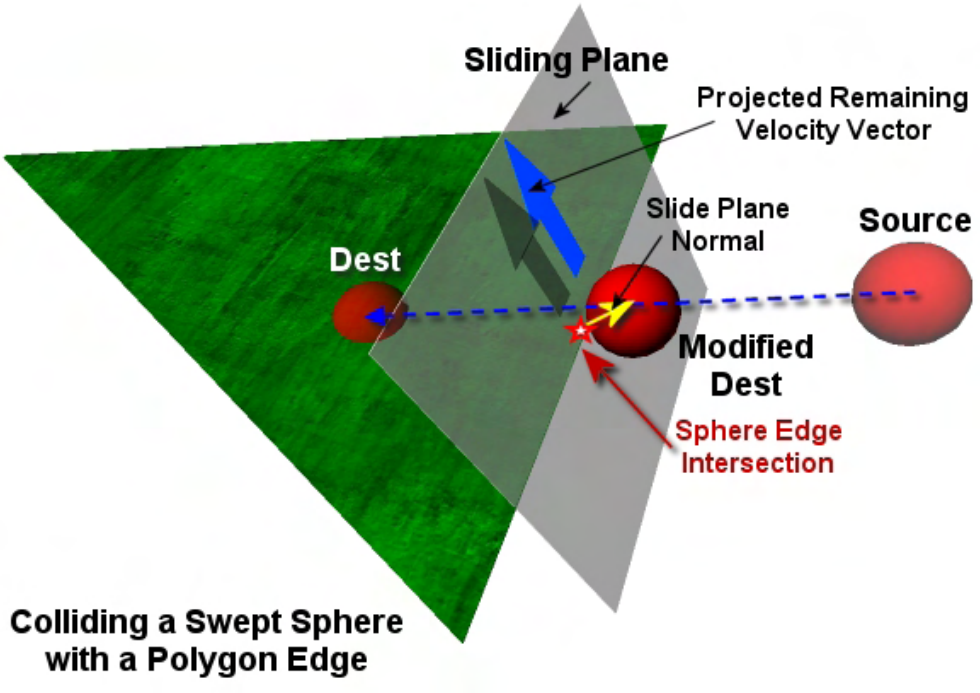
On the bright side, we have developed a collision system that can be easily plugged into our future applications. We will expand the coverage of our CCollision class in the accompanying workbook, where the changes to the application and the CPlayer class will be discussed. We will also discuss all of the CCollision utility member functions, such as those responsible for registering terrains, dynamic objects, and static meshes with the collision database.

The core runtime query functions of the collision system have all been discussed in this chapter, so they will not be covered again. The final EllipsoidIntersectScene, EllipsoidIntersectBuffers and CollideEllipsoid functions just examined are almost identical to the ones used by the CCollision class in Lab Project 12.1. In the lab project we add a few extra lines to efficiently deal with terrains, but this will be explained in the workbook. The SphereIntersectTriangle, SphereIntersectPlane, SphereIntersectPoint,

and SphereIntersectLineSegment intersection routines (called from SphereIntersectTriangle) are also the exact implementations used in Lab Project 12.1. With all the intersection routines covered, this leaves us with a workbook that can concentrate on how geometry is registered with the system and how the collision system is used by our framework.

Your next course of action should be to read the workbook that accompanies this chapter so that you will finally have a full understanding of every aspect of the collision system. As we will be using this collision system in all future lab projects, it is a good idea to make sure you understand it very well.

Workbook Thirteen:
Collision Detection and Response



Lab Project 13.1: The CollisionSystem

In this workbook we will discuss the remaining code in Lab Project 13.1 which was not discussed in the accompanying textbook. It is very important that you read the accompanying textbook before reading this workbook because the intersection routines in the CCollision class will not be discussed again in detail here.

This workbook will focus on the utility functions exposed by the CCollision class which allow an application to easily register various types of static and dynamic objects with the collision system. We will discuss the functions that allow the application to register static meshes with the collision geometry database as well as how to register complete frame hierarchies (contained in a CActor) with the collision system. Furthermore, we will cover the functions that allow the application to register single mesh constructs and entire frame hierarchies as dynamic scene collision objects. This will allow an application to register fully animated actors with the collision system such that collision determination and response between moving entities and animated scene geometry can happen in real time.

Figure 13.1

We will also add a special method tailored for the registration of terrain data that will help keep the memory overhead of the collision geometry database minimized. This will ensure that our CTerrain objects too can benefit from the collision system. The process will involve our collision system building triangle data on the fly using the terrains height map whenever that terrain is queried for collision. This allows us to completely eliminate the need for us to make a copy of every triangle comprising the terrain in order to add it to the static geometry database. Whenever a terrain is queried for intersection, the swept ellipsoid will be transformed into the image space of the terrain's height map. The starting location and the desired destination location of the ellipsoid will then be used to construct an image space bounding box on the height map describing the region of the terrain that could possibly be collided with in this update. This will then be used to build temporary vertex and triangle lists containing only triangles in the area of the terrain that fall within that box (see Figure 13.1). These buffers will then be tested for intersection using our standard tests. The end result is that our collision system will provide detection and response for our CTerrain objects without the need to store copies of every terrain triangle. This makes collision detection against height map based terrains very memory efficient as only a handful of triangles will need to be temporarily stored, and only for the lifetime of the collision test. These triangles will then be discarded.

Our collision system will also be extended to deal with the registration of referenced actors. This means that dynamic objects inside the collision system will not always have their own vertex and triangle lists. We may, for example, have a frame hierarchy loaded into a CActor object that contains 20 meshes. This actor might even be used to model a geometric construct that appears many times in our scene; a street lamp for example. We already know that from a rendering perspective, rather than having to load 20 separate actors into memory, all with the same geometric detail, we can just create 20 CObjects that all point to the same actor but have different world matrices (and possibly animation controllers). As long as we inform the actor of the objects world matrix so that it can update itself before it is rendered, we can have just one actor in memory and render it from in 20 different places in our scene.

We also took this concept a step further earlier in the course when we added support for referenced *animated* actors. To do this we expanded the CObject structure to store both a pointer to the actor it is referencing and a pointer to the animation controller for that reference. Before we render the object, we pass its world matrix into the actor so that it can update all its frame matrices correctly for that reference. We also attach the object's controller to the actor. Attaching the instance controller allowed the actor to update its frame matrices so that they reflect the positions and orientations described by the animation controller for that reference.

We have to consider this idea with respect to our collision database. When we add an actor to our collision database, the CCollision class will traverse the hierarchy and add a new dynamic object to the internal list of dynamic objects for every mesh container found in the hierarchy. For each mesh container we find, we will copy over its vertex and index data into a new dynamic object. Clearly, if we have a single actor that contained 10 meshes and we would like to place it in the scene 100 times in different world space locations, we would not want to create 1000 dynamic objects in our collision system (100*10) that all had their own copies of the same vertex data. To address this, we will allow an actor to be added as a reference to an actor that has previously been registered with the collision system. We know from the textbook discussion that when an actor is registered with the collision system, every mesh in that hierarchy will have a dynamic object created for it and each will be assigned the same object set index. Using the previous example of a 10 mesh hierarchy, when the actor is added for the first time, 10 dynamic objects would be created and added to the collision database. However, each of these dynamic objects would be assigned the same object set ID. This is how the collision system knows that all these objects belong to the same hierarchy or group and need to be updated together.

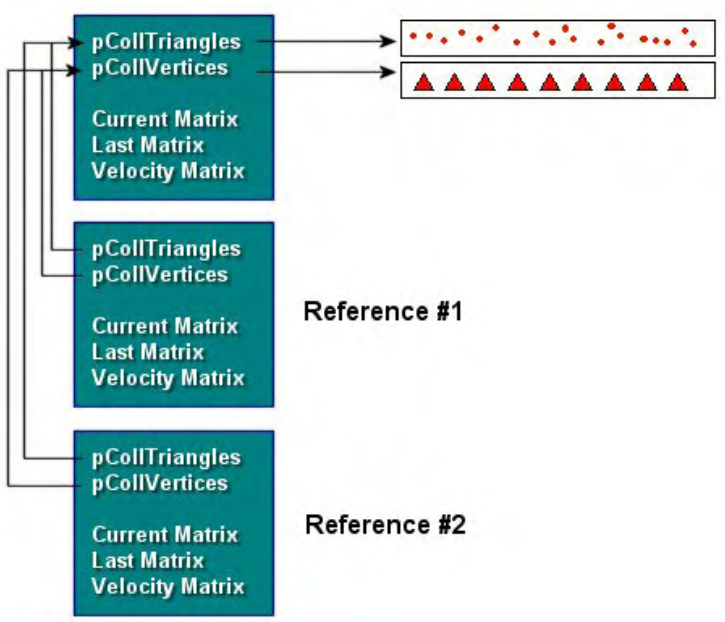


Figure 13.2

The next time we wish to register the same actor with the collision system we can use the CCollision::AddActorReference function. This function will be passed the object set index that was returned by the collision system when the initial actor was added. This function will then loop through the collision systems dynamic object array searching for all dynamic objects that have a matching object set index. It knows that each one it finds belongs to the original actor that is now being referenced. As such, it can just make a new copy of each dynamic object and have the vertex and triangle buffer pointers point to the original dynamic objects buffers.

For example, if we initially add an actor that contains 20 meshes to the collision system, 20 dynamic objects will be created. Each will contain actual vertex and index information from the original meshes in the hierarchy that was registered with the collision system. The registration of this actor would cause

the collision system to issue an object set index back to the application that will have been assigned to each dynamic object created from that actor. If the application subsequently passed this object set index into the CCollision::AddActorReference method, the 20 dynamic objects would be duplicated such that there would now be 40 dynamic objects registered with the collision system. However, the vertex and triangle buffers of the second 20 objects would be pointers to the buffers of the original 20 objects from which they were cloned. Remember from the textbook that a non-referenced dynamic object will contain its geometry data in model space. Since each dynamic object (normal or referenced) contains its own world matrix, the collision system has everything it needs to take that geometry and transform it into world space prior to performing any intersection tests on that dynamic object

We will also examine how the CScene IWF loading functions (e.g., ProcessMeshes and ProcessReference) have been altered so that they can register each mesh, actor, and terrain they create with the collision system. The CScene class owns the CCollision object as a member variable.

Finally, we will discuss how the application (and especially the CPlayer object) uses the collision system in a real-time situation. We will add a better physics model to the movement of our CPlayer so that now concepts like friction and gravity are factored when computing its velocity. The velocity of the CPlayer will ultimately be controlled by how much force is being applied to it. The CGameApp::ProcessInput function will also be altered slightly so that instead of moving the player directly in response to key presses, these now cause an application of force to the player. We then let our simple physics model determine the velocity of the player based on the force being applied, taking into account other factors such as resistant forces (aerodynamic drag, for example).

In this workbook will cover the following:

- Adding geometry registration functions to CCollision
- Adding referenced data to the collision system
- Adding memory efficient terrain registration and collision testing functions for height-map based terrains
- Modifying our IWF and X file loading functions to automate geometry registration with the collision system
- Interfacing the application with our collision system
- Applying a simple Newtonian physics model to our player to accommodate movement that accounts for forces (friction, drag, gravity, etc.)

The CCollision Class

The CCollision class will be a member of our CScene class. It will be charged with the task of determining collisions with scene geometry and generating appropriate responses. The CCollision class has its core intersection functions declared as static members so that they can be used directly by external sources in the event that the full suite of functionality offered by this class is not required by the user. All of the query functions have been discussed in detail in the textbook, but there will be a minor upgrade to the EllipsoidIntersectScene function presented in the textbook to accommodate CTerrain objects that may have been registered with the collision system. So we will have to look at that later.

The CCollision class is declared in CCollision.h and its implementation is in CCollision.cpp. Below we see the CCollision class declaration. We have removed the list of member functions to improve readability, so check the source code for a full list of methods. All we are interested in at the moment are the member variables and structures declared within the class scope. If you have read the textbook, then all of these structures and nearly all of the member variables will be familiar to you. We have added one or two (in bold below) that were not shown in the textbook version of the class to facilitate support for height mapped terrains.

```
class CCollision
{
public:

    //-----------------------------------------------------------------------
    // Public, class specific, Typedefs, structures & enumerators
    //-----------------------------------------------------------------------
    struct CollTriangle
    {
        ULONG       Indices[3];         // Triangle Indices
        D3DXVECTOR3 Normal;             // The cached triangle normal.
        USHORT      SurfaceMaterial;    // The material index of this triangle.
    };

    // Vectors
    typedef std::vector<CollTriangle>   CollTriVector;
    typedef std::vector<D3DXVECTOR3>    CollVertVector;

    struct DynamicObject
    {
        CollTriVector  *pCollTriangles;   // Triangle List
        CollVertVector *pCollVertices;    // vertex List
        D3DXMATRIX     *pCurrentMatrix;   // Pointer to dynamic objects
                                          // application owned external matrix
        D3DXMATRIX      LastMatrix;       // The matrix recorded on the last test
        D3DXMATRIX      VelocityMatrix;   // Describes movement from previous
                                          // position to current position.
        D3DXMATRIX      CollisionMatrix;  // Space in which collision is performed
        long            ObjectSetIndex;   // The index of the set of objects this
                                          // belongs to inside the collision
                                          // systems geometry database.
        bool            IsReference;      // Is this a reference object?
    };

    // Vectors
    typedef std::vector<DynamicObject*> DynamicObjectVector;

    struct CollIntersect
    {
        D3DXVECTOR3     NewCenter;      // The new sphere/ellipsoid centre point
        D3DXVECTOR3     IntersectPoint; // Collision point on surface of ellipsoid
        D3DXVECTOR3     IntersectNormal;// The intersection normal (sliding plane)
        float           Interval;       // The Time of intersection (t value)
        ULONG           TriangleIndex;  // The index of the intersecting triangle
        DynamicObject * pObject;        // A pointer to the dynamic object that has
                                        // been collided with.
    };
```

```
    // Vectors
    typedef std::vector<CTerrain*>      TerrainVector;

    // Constructors & Destructors for This Class.
             CCollision();
    virtual ~CCollision();

 private;

    // Private Variables for This Class.
    CollTriVector       m_CollTriangles;
    CollVertVector      m_CollVertices;
    DynamicObjectVector m_DynamicObjects;

    TerrainVector       m_TerrainObjects;
    USHORT              m_nTriGrowCount;
    USHORT              m_nVertGrowCount;

    USHORT              m_nMaxIntersections;
    USHORT              m_nMaxIterations;
    CollIntersect      *m_pIntersections;

    D3DXMATRIX          m_mtxWorldTransform;
    long                m_nLastObjectSet;
};
```

We have added only three new member variables beyond the version covered in the textbook. Let us briefly discuss them.

TerrainVector m_TerrainObjects
In the last lesson we introduced our CTerrain class, which encapsulated the creation of terrain geometry from height maps. We also added support in our IWF loading code for the GILES™ terrain entity. We can use GILES™ to place terrain entities in our IWF level and have those terrains created and positioned by our CTerrain class at load time. Each CTerrain object contains a height map and the actual geometric data created from that height map.

We will develop a memory efficient way for these terrain objects to be registered with the collision system by using the terrain height map to build only a temporary list of triangles in the region of the swept sphere as and when they are needed. As terrains are usually quite large and are comprised of many triangles, we certainly want to avoid storing a copy of every terrain triangle in the static geometry database. All our collision system will need is an array where it can store CTerrain pointers. The CCollision interface will expose an AddTerrain method which will essentially just add the passed CTerrain pointer to this internal list of terrain objects. That is all it does. No geometry is added to the collision system. When the EllipsoidIntersectScene function is called in the detection step, these CTerrain pointers will be used to access the height map of each terrain. These height maps will then be used to build only triangles that are within the same region as the swept sphere.

This member variable will be used to store any pointers to CTerrain objects that are registered with the collision system. It is of type TerrainVector, which is a typedef for an STL vector of CTerrain pointers.

typedef std::vector<CTerrain*> TerrainVector;

USHORT m_nTriGrowCount
USHORT m_nVertGrowCount
STL vectors are used throughout the collision system for storing vector and triangle data. As you know, STL vectors encapsulate the idea of a dynamic array. The STL vector has methods to easily add and delete elements to/from an array as well as methods to resize an array if we have added elements up to its maximum capacity.

The collision system has a vector for storing static triangle structures and a vector for storing static vertices. Additionally, each dynamic object we create will also have its own vectors to store its model space triangle and vertex data. Finally, with our dynamic terrain mechanism, we will be building triangle data on the fly and adding that triangle and vertex data to temporary vectors so that they can be passed into the intersection methods of the collision system. Suffice to say, that during the registration stages especially, we will be adding a lot of triangles and vertices to these vectors. This is especially true of the collision systems static triangle and vertex buffers, since they will have geometry data cumulatively added to them as multiple files are loaded and multiple meshes registered with the static database.

We never know how much space we are going to need in these vectors until we have loaded and registered all the geometry, so we usually set the initial capacity of these buffers to some initial small value. If we wish to add a vertex or a triangle to one of these vectors but find that it is full, we can simply resize the array and add an additional element on the end to place our new data. While the STL vector has methods to allow us to do that easily, we must remember that underneath the hood an array resize is still happening and they can be quite costly. Usually an array resize results in a totally new memory block being allocated that is large enough to store the new capacity before the data from the old memory block is copied into it. The memory of the original array is then deleted and the variable adjusted to point at the new memory block. Of course, all of this happens behind the scenes, but if we imagine that we wish to add 50,000 vertices to the collision databases static vertex vector, as each one is added, the array would have to be resized so that in the worst case there is only enough space for that new vertex. In this worst case, that would mean that while registering those 50,000 vertices, we have also performed 50,000 memory allocations, 50,000 block memory copies and 50,000 memory de-allocations. Again, that is a worst case scenario as most STL vector implementations are far more efficient than this. Still, we wish to avoid slowing down our registration of geometry too much and would like to try to avoid excessive memory fragmentation.

The solution is simple if we are prepared to spare a little memory. We can instead make the decision that when a vector is full and we wish to add another element to it, we can increase its capacity by a fixed size (e.g., 500). We can now add another 500 vertices before we fill it up and have to increase its capacity again. This would cut down our (worst case) 50,000 resize example down to 100. To be sure, at the end of the loading process, this vector could have a few hundred unused elements at the end of its array. But even this can be corrected after the loading process is complete by resizing the vector so that the capacity of the vector matches its number of contained elements.

The m_nTriGrowCount and m_nVertGrowCount variables contain the resize amount for times when vertices are triangles are being added to a data buffer that is currently full. If you set m_nTriGrowCount to 250, when the m_nCollTriangles vector reaches capacity and more space is needed, it will be resized to accommodate another 250 triangles (even though we might only be adding just one). We have seen all of this logic many times before, so it should be quite familiar to you.

D3DXMATRIX m_mtxWorldTransform

The m_mtxWorldMatrix member is used when adding of triangle data to the static database. When we load meshes from a file, those meshes will typically be defined in model space and accompanied by a world transformation matrix that describes the location of that mesh within the scene. When we are adding such mesh data to our static collision geometry database, we are only interested in the world space vertex positions. Therefore, the vertices of the mesh should be transformed by that matrix prior to being stored in the static collision database. The CCollision class exposes a SetWorldTransform method to set this matrix member to the world transform for the mesh that is about to have its vertex data registered. When the vertices of that mesh are passed into the CCollision::AddIndexedPrimitive method to register them with the static database, the vertices will be transformed by this matrix into world space prior to being added to the static collision geometry list. Thus, application registration of three static model space meshes with the collision system would take on the following form:

```
pCollision->SetWorldTransform(…);
pCollision->AddIndexedPrimitive(…);

pCollision->SetWorldTransform(…);
pCollision->AddIndexedPrimitive(…);

pCollision->SetWorldTransform(…);
pCollision->AddIndexedPrimitive(…);
```

Let us now discuss the functions to the CCollision class that we have not yet seen. We will start by looking at the simple initialization functions (the constructor, etc.) and will then discuss the geometry registration functions over several sections. We will then look at the upgraded EllipsoidIntersectScene function and see how we have added intersection testing for registered CTerrain objects.

CCollision::CCollision()

In our application, we will only need to use one instance of a CCollsion object. To keep things simple, this object will be a member of the CScene class. The only constructor is the default one, which simply initializes the member variables of the collision system at application startup.

```
CCollision::CCollision()
{
    // Setup any required values
    m_nTriGrowCount     = 100;
    m_nVertGrowCount    = 100;
    m_nMaxIntersections = 100;
    m_nMaxIterations    = 30;
    m_nLastObjectSet     = -1;

    // Reset the internal transformation matrix
    D3DXMatrixIdentity( &m_mtxWorldTransform );

    // Allocate memory for intersections
    m_pIntersections    = new CollIntersect[ m_nMaxIntersections ];
}
```

The default STL vector resize quantity is set to 100 for both vertex and triangle buffers. Every time our collision system tries to add another triangle or another vertex to an STL vector that is currently filled to capacity, the vector will have its capacity increased by 100. We also set the size of the intersection buffer to 100. This is the CollIntersect buffer that is used by the EllipsoidIntersectScene function to return collision information back to the response phase. We use a default value of 100 as it is unlikely a moving entity will collide with 100 triangles in a single update. Either way, the collision system is really only interested in the first intersection, so even if the buffer is not large enough to return all intersection information structures back to the response step, it will not adversely effect the collision detection or the response phase. The remaining intersection information is only used to compile a bounding box that is returned to the application. Feel free to change this value if needed.

We also set the maximum number of iterations that will be executed by the collision system to a default of 30 loops. If our CollideEllipsoid cannot deplete the slide vector and calculate the final resting place the of the ellipsoid after 30 tries, the collision detection phase will be called one more time and the first non-intersecting position returned will be used. This should rarely happen, but it allows us to cover ourselves when it does.

There is also a member variable called m_nLastObjectSet which is used by the system to store the last object set index that was issued. This is set to -1 initially because we have not yet added any dynamic objects and our collision system has not yet issued any object set indexes. Every time a new group of dynamic objects is registered with the collision system, this value will be incremented and assigned to each dynamic object in that group.

We also set the collision object's world transformation matrix to an identity matrix. The application can set this to a mesh world matrix when it wishes to register a mesh with collision database.

Finally we allocate the array of CollIntersect structures that will be used as the transport mechanism to return triangle intersection information from the detection phase (the EllipsoidIntersectScene function) back to the response phase (the CollideEllipsoid function).

CCollision::SetTriBufferGrow / SetVertBufferGrow

These two functions are simple utility functions that allow you to alter the amount that the capacity of either a vertex or triangle STL vector is expanded when it becomes full and new data is about to be added.

```
void CCollision::SetTriBufferGrow( USHORT nGrowCount )
{
    // Store grow count
    m_nTriGrowCount = max( 1, nGrowCount );
}
```

```
void CCollision::SetVertBufferGrow( USHORT nGrowCount )
{
    // Store grow count
    m_nTriGrowCount = max( 1, nGrowCount );
}
```

Notice that in each function we do not simply store the passed nGrowCount value in the respective CCollision member variables because we must not ever allow these values to be set to zero. If we did, it would break our collision system. We use these values to make room for the extra elements that are about to be added, so it must be set to at least 1.

CCollision::SetMaxIntersections

This function allows you to alter the size of the buffer that is used to transport triangle collision information from the detection phase to the response phase. It essentially defines how many intersections in a given update we are interested in recording information for. This must be set to a size of at least 1 since the response phase uses the first element in this buffer to extract the new ellipsoid position, intersection point, and collision normal. Any additional elements in this array are collisions with other triangles at the exact same *t* value. The application may or may not wish to know which other triangles were hit and as such, we can adjust the size of this array to fit our needs. The default size is 100, but the application can change this default value by calling this function.

```
void CCollision::SetMaxIntersections( USHORT nMaxIntersections )
{
    // Store max intersections
    m_nMaxIntersections = max( 1, nMaxIntersections );

    // Delete our old buffer
    if ( m_pIntersections ) delete []m_pIntersections;
    m_pIntersections = NULL;

    // Allocate a new buffer
    m_pIntersections = new CollIntersect[ m_nMaxIntersections ];
}
```

Notice how the function will not allow the value of m_nMaxIntersections to be set to a value of less than 1 because the system needs to record at least one collision in order to employ the response step. The function then deletes the previous array and allocates a new array of the requested size to replace it.

Collision::SetMaxIterations

This function allows the application to tailor the maximum number of response iterations that will be executed while trying to resolve the final position of the ellipsoid in a collision update (see earlier discussions). The default is 30 iterations (see constructor).

```
void CCollision::SetMaxIterations( USHORT nMaxIterations )
{
    // Store max iterations
    m_nMaxIterations = max( 1, nMaxIterations );
}
```

Once again, we must have at least a single iteration or our collision detection and response code would never be called at all.

CCollision::SetWorldTranform

This simple function can be used by the application to send a world transformation matrix to the collision system before registering any vertex and triangle data for a mesh that needs to be converted into world space.

```
void CCollision::SetWorldTransform( const D3DXMATRIX& mtxWorld )
{
    // Store the world matrix
    m_mtxWorldTransform = mtxWorld;
}
```

When we load internal meshes exported from GILES™, the vertices are already defined in world space, so we can pass them straight into the AddIndexedPrimtive function and leave this matrix set at identity (default). When this is not the case, we must set this matrix prior to registering each model space mesh.

Geometry Registration Methods

This section will discuss the geometry registration methods exposed by the CCollision interface. These are the functions an application will use to register pre-loaded or procedurally generated geometry with the collision system prior to entering the main game loop.

CCollision::AddBufferData

Many of the registration functions that are exposed to the application by the CCollision class will typically involve adding vertex data and triangle data to STL vectors. Whether these vectors are the ones that store the static vertex and triangle geometry buffers or whether they represent the vertex and triangle geometry buffers for a single dynamic object, we will wrap this functionality in a private generic function that can be called to perform this core task for all registration functions. Now that we are adding specialized terrain support to our collision system, we will need to build triangle data for that terrain on the fly when the terrain is queried for collision. These terrain triangles and vertices will also need to be added to temporary STL vectors so that they too can be passed into the EllipsoidIntersectBuffers function which performs the core intersection processes.

Before we discuss the geometry registration functions exposed by the CCollision class, we will first examine the code to the CCollision::AddBufferData method. This is a generic function that is passed two STL vectors (one stores vertices and another stores triangles) and arrays of vertex and index data that should be added to these STL vectors. This function is called from many places in the collision system to take vertex and index data, create triangle information from it, and then add the triangle data to the two passed buffers. By placing this code in a generic function like this, we can use the same code to add geometry to the static geometry STL vectors, the STL vector of each dynamic object, and the temporary STL vectors that are used at runtime to collect relevant terrain data in response to a collision query.

This function will convert the input data into the format that our vertex and triangle STL vectors require. This will involve constructing triangles from the passed indices and generating the triangle normals. This information can then be stored in a CollTriangle structure and added to the passed triangle buffer. The function is also responsible for recognizing when an STL vector is full and expanding its capacity by the values stored in the m_nTriGrowCount and the m_nVertGrowCount member variables.

Finally, remember that an application will never directly call this function; it is private. It is simply a helper function that is used internally by the collision system to cut down on the amount of redundant code that would otherwise have to be added to each registration function. All of these registration functions need to add geometry to buffers, so there would be no sense in duplicating such code in every function.

Let us look at this function a few sections at a time, starting first with its parameter list.

```
bool CCollision::AddBufferData( CollVertVector& VertBuffer,
                                CollTriVector& TriBuffer,
                                LPVOID Vertices,
                                LPVOID Indices,
                                ULONG VertexCount,
                                ULONG TriCount,
                                ULONG VertexStride,
                                ULONG IndexStride,
                                const D3DXMATRIX& mtxWorld )
{
    ULONG        i, Index1, Index2, Index3, BaseVertex;
    UCHAR       *pVertices = (UCHAR*)Vertices;
    UCHAR       *pIndices  = (UCHAR*)Indices;
    CollTriangle Triangle;
    D3DXVECTOR3  Edge1, Edge2;

    // Validate parameters
    if ( !Vertices || !Indices || !VertexCount || !TriCount ||
          !VertexStride || (IndexStride != 2 && IndexStride != 4) )
       return false;
```

The first two parameters are the STL vectors that are going to receive the vertex and triangle data passed into this function. These buffers may be the static scene buffers or the buffers of a single dynamic object. In this function, we are not concerned with who owns these buffers or what they are for, only with filling them with the passed data.

The third and fourth parameters are void pointers to the vertex and index data that we would like to format into triangle data that our collision system understands. These might be the void pointers returned to the application when it locked the vertex and index buffers of an ID3DXMesh, or just pointers to blocks of system memory that contain vertex and index data. The VertexCount and TriCount parameters instruct the function as to how many vertices and indices are contained in these passed arrays. For example, the VertexCount parameter tells us how many vertices are in the Vertices array and the TriCount parameter tells us how many triplets of indices exist in the Indices array.

The VertexStride parameter tells the function about the size of each vertex (in bytes) in the passed vertex array. Although our function will only be interested in the positional X, Y and Z components of each vertex stored in the first 12 bytes of each vertex structure, each vertex may contain other data that our collision system is not interested (e.g., texture coordinates or a vertex normal). We will need to know the size of each vertex in the function so that we can iterate though each vertex in the array and extract only the information we need.

The IndexStride parameter should contain a value describing the size (in bytes) of each index in the indices array. As an index will be either a WORD or DWORD in size, this value will be either 2 or 4 respectively. We need to know this so that we know how to traverse the array.

The final parameter is a matrix that should be used to transform the vertices in the vertex array (into world space) prior to adding them to the STL vector.

Notice how we cast the vertex and index arrays pointers to local unsigned char pointers (pVertices and pIndices). These pointers will allow us to step through those memory blocks one byte at a time so that we can extract the information we need. We also allocate a single CollTriangle structure that will be used to temporarily build each triangle we are going to add to the CollTriangle STL vector.

Our first real task is to test the current maximum capacity of the passed STL vector that will receive the vertex data. If it is not large enough to contain all the vertices it currently contains (VertBuffer.Size) plus the number of vertices we are about to add (VertexCount) then we will have to grow the vector. We grow the vector by the number of elements stored in the m_nVertGrowCount variable (by default set to 100). Since even this may not be enough, we execute this code in a while loop so the vector will continually have its capacity increased until such a time as it is large enough to accept all the vertices we intend to add.

```
    // Catch template exceptions
    try
    {
        // Grow the vertex buffer if required
        while ( VertBuffer.capacity() < VertBuffer.size() + VertexCount )
        {
            // Reserve extra space
            VertBuffer.reserve( VertBuffer.capacity() + m_nVertGrowCount );

        } // Keep growing until we have enough
```

We also do exactly the same thing for the STL vector that is going to receive the triangle information. We resize its capacity until it is large enough to contain the triangles it already contains plus the ones we wish to add to it in this function.

```
        // Grow the triangle buffer if required
        while ( TriBuffer.capacity() < TriBuffer.size() + TriCount )
        {
            // Reserve extra space
            TriBuffer.reserve( TriBuffer.capacity() + m_nTriGrowCount );

        } // Keep growing until we have enough
```

Before we add any vertices to the vertex vector, we must record the current number of vertices that are stored there. We will need to offset the indices we are about to add by this amount so that when the are placed into the CollTriangle structure and added to the triangle vector, they still reference the (original) vertices that were passed in the accompanying vertex array.

We store the current number of vertices in the vector in a local variable called BaseVertex. When we store the indices in the triangle structure, we will add BaseVertex to each index. This will ensure that the triangle indices we add index into the correct portion of the vertex vector and maintain their relationship with their associated vertices.

```
        // Store the original vertex count before we copy
        BaseVertex = VertBuffer.size();
```

Now we will loop through each vertex in the passed vertex array using the vertex stride to move our byte pointer from the start of a vertex to the next. Our byte pointer pVertices will always point to the start of a vertex in this array. Since we are only interested in the positional 3D vector stored in the first 12 bytes, we can cast this pointer to a D3DXVECTOR3 structure to extract the values into a temporary D3DXVECTOR3. We will then transform this vertex position by the input world transformation matrix before adding the transformed position to the vertex vector (using the vector::push_back method).

```
        // For each vertex passed
        for ( i = 0; i < VertexCount; ++i, pVertices += VertexStride )
        {
            // Transform the vertex
            D3DXVECTOR3 Vertex = *(D3DXVECTOR3*)pVertices;
            D3DXVec3TransformCoord( &Vertex, &Vertex, &mtxWorld );

            // Copy over the vertices
            VertBuffer.push_back( Vertex );

        } // Next Vertex
```

At this point we have successfully transformed all the vertices in the passed array into world space and added them to the passed vertex STL vector. Now we must add the triangles to the passed CollTriangle vector.

We set up a loop to count up to the value of TriCount. Since the data is assumed to represent an indexed triangle list, we know that there should be 3*TriCount indices in the passed array. For each iteration of the loop, we will extract the next three indices in the array and make a triangle out of them. We will also generate the triangle normal before adding the triangle structure to the CollTriangle vector. Our collision system will need to know the triangle normal if the triangle has its interior collided with as this will be used as the slide plane normal.

First we will examine the section of code that extracts the three indices from the array. This may initially look a little strange, but we will explain how it works in a moment.

```
        // Build the triangle data
        for ( i = 0; i < TriCount; ++i )
        {
```

```
            // Retrieve the three indices
            Index1 = ( IndexStride == 2 ? (ULONG)*((USHORT*)pIndices)
                                        : *((ULONG*)pIndices) );
            pIndices += IndexStride;

            Index2 = ( IndexStride == 2 ? (ULONG)*((USHORT*)pIndices)
                                        : *((ULONG*)pIndices) );
            pIndices += IndexStride;

            Index3 = ( IndexStride == 2 ? (ULONG)*((USHORT*)pIndices)
                                        : *((ULONG*)pIndices) );
            pIndices += IndexStride;
```

We currently have a byte sized pointer to the index array, so we know that we are going to have to cast the index pointer to either a USHORT or a ULONG (WORD or DWORD) pointer in order to dereference that pointer and grab a copy of the actual index in the correct format (depending on whether we have been passed 16 or 32-bit indices). However, our CollTriangle structures *always* store indices as ULONGs (32-bit values), so in the case where we have a byte pointer to 16-bit indices, a double cast is necessary.

To clarify, if the index stride is 2, then we have an array of 16-bit (USHORT) indices. When this is the case we must first cast the byte pointer (pIndices) to a pointer of the correct type (USHORT):

```
(USHORT*) pIndices
```

We then want to dereference this pointer to get the actual value of the index pointed to:

```
*((USHORT*)pIndices)
```

We now have a 16-bit value, but our collision system wants it as a 32-bit value so we do an additional cast to a ULONG on the de-referenced 16 bit value.

```
(ULONG)*((USHORT*)pIndices)
```

If the index stride parameter is not set to 2, we have a byte pointer to an array of 32-bit (ULONG) values. When this is the case, we can simply cast the byte pointer to a ULONG pointer and deference the result.

```
*((ULONG*)pIndices)
```

Notice that as we extract the information for each index into the local variables (Index1, Index2, and Index3), we increment the byte pointer by the stride of each index (IndexStride) so that it moves the correct number of bytes forward to be pointing at the start of the next index in the array.

In the next section of code we start to fill out the local CollTriangle structure with the information about the current triangle we are about to add. In this function, we set the surface material index of each triangle to zero as this can be set by a higher level function. The surface material index allows you to assign some arbitrary numerical value to a collision triangle so that your application can determine what it should do if a collision with a triangle using that material occurs. For example, if the triangle has a

toxic material or texture applied, the application may wish to degrade the health of the player after contact is made.

In the following code, we also add the three indices we have fetched from the indices array to the indices array of the CollTriangle structure. As we add each index, we remember to add on BaseVertex to account for the fact that the original indices were relative to the start of the passed vertex array and not necessarily the start of the vertex STL vector.

```
            // Store the details
            Triangle.SurfaceMaterial = 0;
            Triangle.Indices[0]      = Index1 + BaseVertex;
            Triangle.Indices[1]      = Index2 + BaseVertex;
            Triangle.Indices[2]      = Index3 + BaseVertex;
```

We now need to generate a normal for this triangle. This will be used by the collision detection phase as the collision (slide plane) normal when direct hits with the interior of the triangle occur.

In order to calculate the normal of a triangle we need to take the cross product of its two edge vectors. In order to calculate the edge vectors we use our new indices to fetch the triangles vertices from the vertex vector.

```
            // Retrieve the vertices themselves
            D3DXVECTOR3 &v1 = VertBuffer[ Triangle.Indices[0] ];
            D3DXVECTOR3 &v2 = VertBuffer[ Triangle.Indices[1] ];
            D3DXVECTOR3 &v3 = VertBuffer[ Triangle.Indices[2] ];
```

We now create an edge vector from the first vertex to the second vertex and another from the first vertex to the third vertex and normalize the result.

```
            // Calculate and store edge values
            D3DXVec3Normalize( &Edge1, &(v2 - v1) );
            D3DXVec3Normalize( &Edge2, &(v3 - v1) );
```

We then perform a safety check to make sure that we have not been passed a degenerate triangle. We take the dot product of the two edge vectors we just calculated. If the absolute result is equal to 1.0 (with tolerance) we know that the three vertices of the triangle exist on the same line. This triangle will provide no benefit in our collision system as it has zero volume and cannot be collided with. When this is the case, we skip to the next iteration of the triangle generation loop and avoid adding the current triangle to the triangle vector.

```
            // Skip if this is a degenerate triangle, we don't want it in our set
            float fDot = D3DXVec3Dot( &Edge1, &Edge2 );
            if ( fabsf(fDot) >= (1.0f - 1e-5f) ) continue;
```

We now take the cross product of the two edge vectors and normalize the result to get our triangle normal. We store the result in the CollTriangle structure before finally adding the triangle structure to the back of the CollTriangle vector.

```
            // Generate the triangle normal
            D3DXVec3Cross( &Triangle.Normal, &Edge1, &Edge2 );
            D3DXVec3Normalize( &Triangle.Normal, &Triangle.Normal );

            // Store the triangle in our database
            TriBuffer.push_back( Triangle );

        } // Next Triangle

    } // End Try Block

    catch ( ... )
    {
        // Just fail on any exception.
        return false;

    } // End Catch Block


    // Success!
    return true;
}
```

CCollision::AddIndexedPrimitive

This is a public method that can be used to register mesh vertex and index data with the collision system. Lab Project 13.1 uses this function when loading the internal meshes from an IWF file. It will take the loaded vertex and index lists and pass them in along with information about the size of the vertex and index structures. This function is used to register mesh data with the **static** scene geometry database.

AddIndexedPrimitive is just a wrapper around the previous function. You should take note of the variables that are passed by this function to the AddBufferData method. For the first and second parameters it passes in the m_CollVertices and m_CollTriangles member variables. These are the STL vectors used by the CCollision class to contain all its static vertex and triangle geometry. In other words, the AddBufferData function will add the passed geometry data to the static buffers of the collision system. Notice also that the member variable m_mtxWorldTranform is passed in as the final parameter. This is the collision system's current world transformation matrix that will be applied to the static geometry being registered. Now you can see how setting this matrix (via the SetWorldTransform method) before registering a mesh with the collision system transforms the vertices of that mesh into world space before adding them to the static database.

```
bool CCollision::AddIndexedPrimitive( LPVOID Vertices,
                                      LPVOID Indices,
                                      ULONG VertexCount,
                                      ULONG TriCount,
                                      ULONG VertexStride,
                                      ULONG IndexStride )
{
    // Add to the standard buffer
    return AddBufferData(     m_CollVertices,
                              m_CollTriangles,
```

```
                                Vertices,
                                Indices,
                                VertexCount,
                                TriCount,
                                VertexStride,
                                IndexStride,
                                m_mtxWorldTransform );
}
```

CCollision::AddIndexedPrimitive (Overloaded)

This next method overloads the previous one and allows you to specify a numerical material index that will be assigned to each triangle. For example, if you wanted every triangle of this a mesh that was registered with the collision system to have a material index that matches its subset ID in the original D3DX mesh, you would call this AddIndexedPrimitive version multiple times for the mesh, once for each subset. In each call, you would pass only the vertices and indices of the current subset you are rendering and the subset ID as the material index.

```
bool CCollision::AddIndexedPrimitive( LPVOID Vertices,
                                      LPVOID Indices,
                                      ULONG VertexCount,
                                      ULONG TriCount,
                                      ULONG VertexStride,
                                      ULONG IndexStride,
                                      USHORT MaterialIndex )
{
    ULONG i, BaseTriCount;

    // Store the previous triangle count
    BaseTriCount = m_CollTriangles.size();

    // Add to the standard buffer
    if ( !AddBufferData( m_CollVertices,
                         m_CollTriangles,
                         Vertices,
                         Indices,
                         VertexCount,
                         TriCount,
                         VertexStride,
                         IndexStride,
                         m_mtxWorldTransform ) ) return false;

    // Loop through and assign the specified material ID to all triangles
    for ( i = BaseTriCount; i < m_CollTriangles.size(); ++i )
    {
        // Assign to triangle
        m_CollTriangles[i].SurfaceMaterial = MaterialIndex;
    }

    // Success
    return true;
}
```

The above function first records how many triangles are in the scene's static triangle vector before it adds the new buffer data, and stores the result in the BaseTriCount local variable. The AddBufferData function is then called as before to add the vertex and triangle data to the collision system's static geometry database. When the function returns, BaseTriCount will contain the index of the first triangle that was added to the buffer in this call. We can then start a loop through every triangle from the first new one that was added to the end of the triangle buffer. This is the range of triangles that were just added. We set the SurfaceMaterial member of each triangle in this range to the material index passed into the function by the caller.

Collision::AddTerrain

Although we have not yet explained exactly how our collision detection system will dynamically generate triangle data from terrain height maps for collision purposes, we know that it will need access to the CTerrain objects which the application intends to use as collision geometry. The CCollision class stores its registered terrains as a simple vector of CTerrain pointers.

This function is used by application to register CTerrain objects with the static collision database. This function is extremely simple since the collision system will use the CTerrain object pointer to generate the actual collision geometry during the intersection tests. All we need to do in this function is add a pointer to the passed terrain to the collision system's CTerrain pointer vector. We must also call the CTerrain->AddRef method to let the terrain object know that another object has a pointer to its interface. Remember, we recently added the COM reference counting mechanism to the CTerrain class and it must be used (otherwise the terrain object could delete itself from memory prematurely).

```
bool CCollision::AddTerrain( CTerrain * pTerrain )
{
    // Validate parameters
    if ( !pTerrain ) return false;

    // Catch Exceptions
    try
    {
        // We own a reference, make sure it's not deleted.
        pTerrain->AddRef();

        // Store the terrain object
        m_TerrainObjects.push_back( pTerrain );

    } // End Try Block

    catch ( ... )
    {
        // Simply bail
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

CCollision::AddDynamicObject

In the textbook we discussed how dynamic objects would be integrated into the collision system. In terms of our collision system, a dynamic object is a scene object that cannot be stored in the usual static database because its world space position/orientation can change at any moment. A simple example is the mesh of a sliding door. We would want the player to be able to collide with this door so that he/she is not allowed access to a certain area when the door is closed. The door mesh cannot be transformed into world space during registration because the application may wish to alter its world matrix in response to a game event.

When we register a dynamic object with the collision system, a new DynamicObject structure is allocated and added to the collision system's dynamic object STL vector. A dynamic object stores its own model space vertex and triangle data and a pointer to the matrix that the application can update. Whenever the application updates the world matrix for a dynamic object, it should call the CCollision::ObjectSetUpdated method to allow the collision system to recalculate the velocity matrix and collision matrix used by the intersection functions. This was all discussed in the textbook, so we will not review that material here.

This particular function should be used if the caller would like to register a single mesh as a dynamic object. We will not be using this function in Lab Project 13.1 as all the dynamic objects we register will be actors containing complete frame hierarchies of meshes. There are separate functions to aid in the registration of actors.

An application that would like to register a single mesh object with the collision system as a dynamic object should pass this function the model space vertex and index data and a pointer to the application owned world matrix for the dynamic object. A copy if this pointer will be cached in the dynamic object structure so that both the collision system and the application have direct access to it. The application will be responsible for updating the world matrix and the collision system will be able to work with those updates.

Before we look at the code remember that whenever a dynamic object is added to the collision database, it is assigned a collision ID which is referred to as an object set index. This is like the handle for the object within the collision system and is the means by which the application can inform the collision system that the matrix of that object has been updated. Because we may want to add multiple dynamic objects as part of the same object group (they share the same object set index) we have a final boolean parameter to this function. If set to true, the CCollision::m_nLastObjectSet member will be increased, thus creating a new object group assigned to the dynamic object. If this boolean is set to false then the m_nLastObjectSet index will not be increased and the object will be assigned the same index as a previous object that has been registered. This allows us to assign multiple dynamic objects the same ID, much like when an actor is registered with the collision system. If we assigned multiple objects the same ID, we can reference and update all those dynamic objects as a single group using a single collision index. If you want each dynamic object you register to have its own object set index (most often the case) then you should pass true as this parameter.

When this function is called to add the first dynamic object to the collision system, the current value of m_nLastObjectSet will be set to -1, which is not a valid object set index. So you should never pass false

to this function when registering the first dynamic object. Essentially, you are stating that you would like the dynamic object to be added to a previously created group, which makes no sense in the case of the first dynamic object when no groups have yet been created. Just in case, the function will force this boolean to true if this is the first dynamic object that is being added to the system.

The first section of code creates an identity matrix. You will why this is the case in a moment. The function tests to see if the current value of m_nLastObjectSet is smaller than zero and if so, then we know this is the first dynamic object that has been added to the system and the NewObjectSet Boolean is forced to true. The value of the Boolean parameter is then checked and if set to true, the current value of the member variable m_nLastObjectSet is incremented to provide a new object set index for this object. The function then allocates a new DynamicObject structure and initializes it to zero for safety. We then allocate the vertex and triangle STL vectors which will be pointed at by the dynamic object structure and used to contain the model space geometry of the dynamic object's mesh

```
long CCollision::AddDynamicObject(  LPVOID Vertices,
                                    LPVOID Indices,
                                    ULONG VertexCount,
                                    ULONG TriCount,
                                    ULONG VertexStride,
                                    ULONG IndexStride,
                                    D3DXMATRIX * pMatrix,
                                    bool bNewObjectSet /* = true */ )
{
    D3DXMATRIX      mtxIdentity;
    DynamicObject * pObject = NULL;

    // Reset identity matrix
    D3DXMatrixIdentity( &mtxIdentity );

    // Ensure that they aren't doing something naughty
    if ( m_nLastObjectSet < 0 ) bNewObjectSet = true;

    // We have used another object set index.
    if ( !bNewObjectSet ) m_nLastObjectSet++;

    try
    {
        // Allocate a new dynamic object instance
        pObject = new DynamicObject;
        if ( !pObject ) throw 0;

        // Clear the structure
        ZeroMemory( pObject, sizeof(DynamicObject) );

        // Allocate an empty vector for the buffer data
        pObject->pCollVertices  = new CollVertVector;
        if ( !pObject->pCollVertices ) throw 0;

       pObject->pCollTriangles = new CollTriVector;
        if ( !pObject->pCollTriangles ) throw 0;
```

At this point we have a new dynamic object which also has pointers to two new (currently empty) STL vectors for the model space the vertices and triangles of the mesh. We set the dynamic object's matrix

pointer to point at the matrix passed by the application. This will describe the current world space transformation of the dynamic object at all times. We also copy the values of this matrix into the LastMatrix and CollisionMatrix members of the structure since the object is currently stationary. These values will be set correctly the first time the CCollision::ObjectSetUpdated method is called prior to any collision tests.

```
        // Store the matrices well need for
        pObject->pCurrentMatrix  = pMatrix;
        pObject->LastMatrix      = *pMatrix;
        pObject->CollisionMatrix = *pMatrix;
        pObject->ObjectSetIndex  = m_nLastObjectSet;
        pObject->IsReference     = false;
```

Notice in the above code that we then assign the value of the m_nLastObjectSet member variable as the dynamic object's ID. If the Boolean parameter was set to false, this will not have been incremented and the object will be assigned the same ID as the last dynamic object that was created (i.e., adding this dynamic object to a pre-existing object set). If the Boolean parameter was set to true, then the value would have been incremented, creating a new and currently unique ID, making this object the first to be added to this new object set. We also set the reference member of the structure to false since this is not a referenced dynamic object (it has its own vertex and triangle data).

Our next task is to format the passed model space vertices and indices of the dynamic object's mesh and add them to its STL geometry vectors. This is no problem since we can once again use our AddBufferData function for this task.

```
        // Add to the dynamic objects database
        if ( !AddBufferData(  *pObject->pCollVertices,
                              *pObject->pCollTriangles,
                              Vertices,
                              Indices,
                              VertexCount,
                              TriCount,
                              VertexStride,
                              IndexStride,
                              mtxIdentity ) ) throw 0;

        // Store the dynamic object
        m_DynamicObjects.push_back( pObject );

    } // End try block
```

The first and second parameters are the dynamic object's geometry vectors to be filled with the formatted data. The final parameter is the identity matrix we created at the head of the function. We do this because we know that the AddBufferData function will transform all the vertices we pass it by this matrix to transform them into world space. In the case of a dynamic object, we want the vertices to remain in model space since they will be transformed into world space on fly by the collision detection routines. After the AddBufferData function returns, our dynamic object structure will be fully populated and its geometry vectors will contain all the relevant model space vertices and triangles. As a final step, we then add this dynamic object structure to the collision system's dynamic object array.

If anything went wrong in the process and an exception was thrown, we free any memory we may have allocated in the catch block before returning an error value of -1.

```
    catch (...)
    {
        // Release the object if it got created
        if ( pObject )
        {
            if ( pObject->pCollVertices  ) delete pObject->pCollVertices;
            if ( pObject->pCollTriangles ) delete pObject->pCollTriangles;
            delete pObject;

        } // End if object exists

        // Return failure
        return -1;

    } // End Catch Block
```

If we get this far then everything worked and we return the object set index that was assigned to the dynamic object. Our application will store the object set indices of each dynamic object we register in the CObject structure that owns the mesh. When we update the matrix of a CObject, we can then use this object set index to notify the collision system that the matrices of the dynamic object need to be re-cached using the CCollision::ObjectSetUpated method. This was discussed in the textbook.

```
    // Return the object index
    return m_nLastObjectSet;
}
```

CCollision::AddActor

Adding single meshes as dynamic objects is certainly useful, but most of our previous demos have used our CActor class. As we are by now intimately aware, our CActor class encapsulates the loading, rendering and animation of complete frame hierarchies that may contain multiple meshes. We will definitely want our collision system to expose a way to also register these multi-mesh constructs with our collision system.

An actor that has no animation data or that is not intended to be moved throughout the scene is essentially a static actor. Registering it with the collision system will mean storing the triangles of each mesh in that hierarchy in the static collision database. This is not very difficult. We just have to traverse the hierarchy searching for mesh containers and for each one container we find, we will lock its vertex and index buffers and transform the vertices into world space using the mesh container's absolute frame matrix (not the parent relative matrix). This does mean that we will also need to be passed the current world matrix for the entire actor as it will influence the world matrices stored at each frame in the hierarchy. We will need to update the hierarchy's absolute matrices using the passed world matrix as we traverse the hierarchy. This is so we know that all absolute frame matrices in the hierarchy are correctly set to contain the world space transforms for each mesh container before using it to transform their

vertices. For each mesh container, we will transform the vertices into world space and then use the AddBufferData function to add this geometry to the static database.

If the actor contains animation or if the application intends to animate the position of the actor within the scene, then we will also need a means to register an actor as a dynamic object, or more correctly, a dynamic object set. Once again, this is not so difficult. If the actor is to be added as a dynamic object set, we will need a function that will traverse the hierarchy searching for mesh containers. For each mesh container it finds it will allocate a new DynamicObject structure and add it to the collision system's dynamic object array. We will lock the vertex and index buffers of each mesh and copy the *model space* vertex and index data into the dynamic object's geometry arrays (using AddBufferData). We will assign each dynamic object we create from a given actor the same object set ID. We know when the actor is animated and all its frame matrices are updated, calling the ObjectSetUpdated method and passing this single ID will result in all dynamic objects created from that actor having their matrices re-cached. Each dynamic object that is created from an actor will also store (as its matrix pointer), a pointer to the absolute matrix in the mesh container's owner frame. This will always store the world space transformation of the mesh whether an animation is playing or not.

So we need two strategies for registering actors. We will wrap both techniques in a single function that uses its parameters to decide whether the actor should be added as a dynamic object set or as static geometry. We pass a Boolean parameter to this function called Static which, if set to true, will cause the meshes contained inside the actor to be registered statically after having been transformed into world space. Otherwise, each mesh in the hierarchy will have a dynamic object created for it and added to the collision system's dynamic object array. The functions used to actually perform the registration process are private functions called AddStaticFrameHierarchy and AddDynamicFrameHierarchy and are for the registration of static and dynamic actors, respectively.

The wrapper function that our application calls is called AddActor and accepts three parameters. The first is a pointer to the CActor that we would like to register. The second is a matrix describing the placement of the actor within the scene (i.e., the root frame world matrix). The third parameter is the Boolean that was previously discussed that indicates whether the actor's geometry should be registered as static or dynamic triangle data.

```
long CCollision::AddActor(    const CActor * pActor,
                              const D3DXMATRIX& mtxWorld,
                              bool Static /*= true*/ )
{
    // Validate parameters
    if ( !pActor ) return -1;

    // Retrieve the root frame
    D3DXFRAME * pFrame = pActor->GetRootFrame();

    // If there is no root frame, return.
    if ( !pFrame ) return -1;

    // Add as static or dynamic geometry?
    if ( Static )
    {
        if ( !AddStaticFrameHierarchy( pFrame, mtxWorld ) ) return -1;
```

```
    } // End if static
    else
    {
        if ( !AddDynamicFrameHierarchy( pFrame, mtxWorld ) ) return -1;

    } // End if dynamic

    // We have used another object set index.
    m_nLastObjectSet++;

    // Return the object index
    return m_nLastObjectSet;

}
```

Notice that before we call the AddStaticFrameHierarchy or AddDynamicFrameHierarchy functions, we fetch the root frame of the actor to pass in as a parameter. These functions are recursive and will start at the root frame of the hierarchy and traverse to the very bottom searching for mesh containers.

It may seem odd that we increment the m_nLastObjectSet member variable at the end of the function rather than before the recursive functions are called. This is because in the actual recursive functions, this value will be incremented by assigning an object set index of m_nLastObjectSet+1 to each dynamic object we create. In other words, if the last dynamic object we added was issued an ID of 10, all the dynamic objects created by these recursive functions would be assigned an ID of 10+1=11. Remember, every dynamic object created from an actor will be assigned the same object set index and are therefore considered to belong to the same object group/set. When the recursive functions return and the dynamic objects have all been created, we then increment the m_nLastObjectSet variable so it now correctly describes the last object set index that was used. In this example, that would be a value of 11.

Of course, this function does not really do a whole lot by itself since the registration processes for both dynamic and static actors are buried away in the helper functions AddStaticFrameHierarchy and AddDynamicFrameHierarchy. Let us now have a look at these functions, starting first with the function that adds a non-animating actor to the static geometry database.

CCollision::AddStaticFrameHierarchy

This function is called by the AddActor method to register a frame hierarchy and all its contained mesh data with the static geometry database. The function recurses until all mesh containers have been found and their geometry added.

The first parameter is a pointer to the current frame that is being visited. When first called by the AddActor method, this will be a pointer to the root frame of the hierarchy. The second parameter is the world matrix that this frame's parent relative matrix is relative to. When this matrix is combined with the parent relative matrix stored at the frame, we will have the absolute world transformation matrix for the frame and the world matrix for any mesh container that it may contain. This same matrix is also passed along to each sibling frame if any should exist since all sibling frames share the same frame of reference. We can think of this matrix as being the absolute world matrix of the parent frame which,

when combined with the parent relative matrix, will provide the absolute matrix of the current frame. This matrix, once used to transform the vertices of any child mesh containers of this frame, is then passed down to the children of this frame. The matrix parameter passed into this function, when called for the root frame (from AddActor), will be the world matrix of the actor itself.

```
bool CCollision::AddStaticFrameHierarchy( D3DXFRAME * pFrame,
                                          const D3DXMATRIX& mtxWorld )
{
    D3DXMESHCONTAINER * pMeshContainer = NULL;
    LPD3DXBASEMESH      pMesh          = NULL;
    LPVOID              pVertices      = NULL;
    LPVOID              pIndices       = NULL;
    ULONG               nVertexStride, nIndexStride;
    D3DXMATRIX          mtxFrame;

    // Validate parameters (just skip)
    if ( !pFrame ) return true;

    // Combine the matrix for this frame
    D3DXMatrixMultiply( &mtxFrame, &pFrame->TransformationMatrix, &mtxWorld );

    // Retrieve the mesh container
    pMeshContainer = pFrame->pMeshContainer;
```

In the first section of the code (shown above) we first test that the frame pointer passed into the function is valid; if not, we return immediately. We then combine the passed matrix with the parent relative matrix of the current frame to generate the absolute world matrix for this frame. This is the world matrix that describes the world space placement of any meshes that are owned by this frame.

We then grab a copy of the frame's pMeshContainer pointer and store it in a local variable (pMeshContainer) for ease of access. If this pointer is NULL then there are no meshes attached to this frame and we have no geometry to add. We can just jump straight to the bottom of the function where we traverse into the child and sibling frames.

> **Note:** We are not interested in storing skinned meshes since our collision system will not directly support skinned geometry. To use skinned meshes as colliders, a good approach is to register either a geometric bounding volume that encapsulates the skinned structure or a series of low polygon meshes that bound the object and animate with it. This produces very good results with little overhead.

The next section of code loops through each mesh container attached to this frame (remember that a frame may have multiple mesh containers arranged in a linked list). Each iteration of the loop will process a single mesh in the list. If the mesh container pointer is NULL, then there are no meshes attached to this frame and the loop is never executed. Once inside the loop, if the mesh container does not have a NULL ID3DXSkinInfo pointer, then we know it contains skinning information and we skip to the next mesh in the list (see note above).

```
    // Keep going until we run out of containers (or there were none to begin with)
    for ( ; pMeshContainer != NULL;
```

```
            pMeshContainer = pMeshContainer->pNextMeshContainer )
    {
        // Skip if this is a skin container
        if ( pMeshContainer->pSkinInfo ) continue;

        // Attempt to retrieve standard mesh
        pMesh = (LPD3DXBASEMESH)pMeshContainer->MeshData.pMesh;

        // Attempt to retrieve progressive mesh if standard mesh is unavailable
        if ( !pMesh ) pMesh = (LPD3DXBASEMESH)pMeshContainer->MeshData.pPMesh;
```

After we have made sure that the current mesh container does not contain skinning information, we then retrieve a pointer to the ID3DXMesh stored in the mesh container (via the D3DXMESHDATA member pMesh field). If this pointer is NULL then it may mean that a progressive mesh is stored here instead of a regular ID3DXMesh. When this is the case, we assign the local pMesh pointer to the pPMesh member of the container's D3DXMESHDATA structure instead. At this point, we hopefully have either a pointer to a progressive mesh or a regular mesh and we can continue. If pMesh is still NULL, then it contains a mesh data format that our system does not currently support (e.g., a patch mesh).

Now that we have a pointer to the mesh, we can use its interface to inquire about the stride of the vertices contained within. We can also use the ID3DXMesh->GetOptions method to determine whether the indices of the mesh are 16 or 32-bits wide. Notice in the following code that if the D3DXMESH_32BIT flag is set, we set the nIndexStride variable to 4 (bytes); otherwise we set it to 2. We need these pieces of information when we feed the vertices and indices of the mesh into the AddBufferData function.

```
        // If we have a mesh to process
        if ( pMesh )
        {
            try
            {
                // Retrieve the stride values
                nVertexStride = pMesh->GetNumBytesPerVertex();
                nIndexStride  = (pMesh->GetOptions() & D3DXMESH_32BIT) ? 4 : 2;
```

Now that we know how big each vertex and index is, we lock the vertex and index buffers of the mesh. The pointers to the vertex and index data are returned in the pVertices and pIndices local variables. Once we have these pointers, we can pass them straight into the AddBufferData function along with the vertex stride and index stride information, as shown below.

```
                // Retrieve the vertex buffer
                if ( FAILED(pMesh->LockVertexBuffer
                                    ( D3DLOCK_READONLY, &pVertices ) ) )
                  throw 0;

                if ( FAILED(pMesh->LockIndexBuffer
                                    ( D3DLOCK_READONLY, &pIndices ) ) )
                  throw 0;


                // Add to static database
```

```
                if ( !AddBufferData(      m_CollVertices,
                                          m_CollTriangles,
                                          pVertices,
                                          pIndices,
                                          pMesh->GetNumVertices(),
                                          pMesh->GetNumFaces(),
                                          nVertexStride,
                                          nIndexStride,
                                          mtxFrame ) ) throw 0;

                // Unlock resources
                pMesh->UnlockVertexBuffer();
                pVertices = NULL;
                pMesh->UnlockIndexBuffer();
                pIndices = NULL;

            } // End try block
```

There are a few important things to note about the AddBufferData call. First, the first two parameters we are passing in are the STL vectors of the CCollision class that contain the static vertex and geometry data. We are also using the methods of the ID3DXMesh interface to pass in the number of vertices and triangles pointed at by the pVertices and pIndices pointers, respectively. Perhaps most important is the matrix we pass in as the final parameter. This is the combined matrix of the current frame which describes the world space transformation of any meshes attached directly to this frame. The AddBufferData function will use this matrix to transform the vertices of the mesh into world space prior to adding them to the static geometry database.

After the AddBufferData function returns, the current mesh has been added to the static geometry database and we can unlock the vertex and index buffers. Our task is complete for this particular mesh.

If an exception is raised for whatever reason, the catch block below will be executed. It simply forces the unlocking of the vertex and index buffers if they were locked when the exception occurred.

```
            catch (...)
            {
                // Unlock resources
                if ( pVertices ) pMesh->UnlockVertexBuffer();
                if ( pIndices  ) pMesh->UnlockIndexBuffer();

                // Return fatal error
                return false;

            } // End catch block

        } // End if mesh exists

    } // Next Container
```

That concludes the mesh loop of this function. The above sections of code will be executed for every mesh container that is a child of the current frame being visited by our recursive function. Notice at the

top of the loop, we move along the linked list of the mesh container by setting the mesh container pointer to point at its pNextMeshContainer member with each iteration.

If we get to this point, we have processed all the meshes attached to this frame and have added them to the static geometry database. We are now ready to continue our recursive journey down the hierarchy. First we test to see if the current frame has a sibling list. If it has, then we will traverse into that list. Notice that when the function recursively calls itself in order to visit the sibling(s), it is passed the same matrix that was passed into this instance of the function. Remember, this is the combined absolute world matrix of the parent frame. Since all siblings share the same parent frame of reference, this same matrix must be passed along to each of them so that they too can combine it with their parent-relative matrices to generate their own world transforms.

```
    // Process children and siblings
    if ( pFrame->pFrameSibling )
    {
        if ( !AddStaticFrameHierarchy( pFrame->pFrameSibling, mtxWorld ) )
            return false;

    } // End if there is a sibling
```

We have now visited our sibling frames and entered their meshes into the static database. Now we must continue down to the next level of the hierarchy and visit any child frames. When the function steps into the child frame, it passes the absolute world matrix of the current frame as the matrix parameter, not the matrix that was passed into this instance of the function. We have had enough exposure to hierarchy traversals at this point to know why this is the case.

```
    if ( pFrame->pFrameFirstChild )
    {
        if ( !AddStaticFrameHierarchy( pFrame->pFrameFirstChild, mtxFrame ) )
             return false;

    } // End if there is a child

    // Success!
    return true;
}
```

That is all there is to adding an entire frame hierarchy to our static database. It is a simple recursive procedure that searches a hierarchy for meshes and adds them to the static geometry vectors.

CCollision::AddDynamicFrameHierarchy

This function is called by the AddActor method if its Boolean parameter was set to false, indicating that this is not a static actor. When this is the case a recursive procedure will be executed, much like the AddStaticFrameHierarchy function (and identical in many respects). The differences between this function and previous function occur when a mesh container is found. This time, its geometry is not transformed into world space and added to the static geometry array. Instead, a new dynamic object is created and the model space vertices are copied into the geometry arrays of the dynamic object. Also, a

pointer to the absolute matrix of the parent frame is stored inside the dynamic object also so that any changes to the hierarchy (via an animation update or an update of the position of the actor by the application), is immediately available to the collision system via this pointer. Each dynamic object is also assigned the same object set index. As the ID issued to the last dynamic object group added to the collision system will be currently stored in m_nObjectSetIndex, we can add one to this value and assign this new index to each dynamic object created from this hierarchy. As we saw when we discussed the AddActor method, the actual value of m_nObjectSetIndex in incremented when the AddDynamicFrameHierarchy function returns so that m_nObjectSetIndex is updated to store the ID we have just assigned to each dynamic object in the hierarchy.

Let us look at the code a section at a time. Note that a lot of this code is duplicated from the previous function we have just discussed, so we will move a little faster.

This function is called from AddActor and is passed the root frame of the hierarchy and the world matrix of the actor. This is the matrix describing the current position of the actor in the scene. We combine it with the parent relative matrix of the current frame being visited (initially the root) to generate the absolute world transformation matrix of the frame we are visiting. This is the world matrix of any meshes assigned to this frame.

```
bool CCollision::AddDynamicFrameHierarchy( D3DXFRAME * pFrame,
                                           const D3DXMATRIX& mtxWorld )
{
    D3DXMESHCONTAINER * pMeshContainer = NULL;
    LPD3DXBASEMESH      pMesh          = NULL;
    LPVOID              pVertices      = NULL;
    LPVOID              pIndices       = NULL;
    ULONG               nVertexStride, nIndexStride;
    D3DXMATRIX          mtxFrame, mtxIdentity;

    // Validate parameters (just skip)
    if ( !pFrame ) return true;

    // Combine the matrix for this frame
    D3DXMatrixMultiply( &mtxFrame, &pFrame->TransformationMatrix, &mtxWorld );

    // Store identity
    D3DXMatrixIdentity( &mtxIdentity );
```

Notice how we also set up an identity matrix. This will be passed into the AddBufferData function so that it does not transform the vertices of the mesh and instead copies the model space vertices straight into the geometry arrays of the dynamic object.

The next section of code is the same as the last. We set up a loop to traverse the linked list of mesh containers that may exist at this frame and skip any meshes in the list that are skins.

```
    // Retrieve the mesh container
    pMeshContainer = pFrame->pMeshContainer;

    // Keep going until we run out of containers (
    // or there were none to begin with)
```

```
    for( ;  pMeshContainer != NULL;
            pMeshContainer = pMeshContainer->pNextMeshContainer )
    {
        // Skip if this is a skin container
        if ( pMeshContainer->pSkinInfo ) continue;

        // Attempt to retrieve standard mesh
        pMesh = (LPD3DXBASEMESH)pMeshContainer->MeshData.pMesh;

        // Attempt to retrieve progressive mesh if standard mesh is unavailable
        if ( !pMesh ) pMesh = (LPD3DXBASEMESH)pMeshContainer->MeshData.pPMesh;
```

The next section of code is executed if we have a valid mesh object. It once again retrieves the stride of both the indices and the vertices and locks the vertex and index buffers.

```
        // If we have a mesh to process
        if ( pMesh )
        {
            DynamicObject * pObject = NULL;

            try
            {
                // Retrieve the stride values
                nVertexStride = pMesh->GetNumBytesPerVertex();
                nIndexStride  = (pMesh->GetOptions() & D3DXMESH_32BIT) ? 4 : 2;

                // Retrieve the vertex buffer
                if ( FAILED(pMesh->LockVertexBuffer(  D3DLOCK_READONLY,
                                                      &pVertices ) ) ) throw 0;


                if ( FAILED(pMesh->LockIndexBuffer( D3DLOCK_READONLY,
                                                    &pIndices ) ) ) throw 0;
```

We do not wish to add this geometry to the static arrays of the collision system but instead want to create a new dynamic object from this mesh. Thus, we first allocate a new dynamic object, initialize its memory to zero, and then allocate the two STL vectors that the dynamic object will use for its vertex and triangle data.

```
                // Allocate a new dynamic object instance
                pObject = new DynamicObject;
                if ( !pObject ) throw 0;

                // Clear the structure
                ZeroMemory( pObject, sizeof(DynamicObject) );

                // Allocate an empty vector for the buffer data
                pObject->pCollVertices  = new CollVertVector;
                if ( !pObject->pCollVertices ) throw 0;

                pObject->pCollTriangles = new CollTriVector;
                if ( !pObject->pCollTriangles ) throw 0;
```

At this point, our dynamic object has its pCollVertices and pCollTriangles members pointed at the newly allocated empty vectors.

In the next section of code we will assign the pCurrentMatrix member of the dynamic object to point at the frame's combined matrix. This is the matrix that will contain the world space transform for this frame (and any of its attached meshes) when the actor is updated. We do not care what is currently stored in this matrix as we will not use it until the ObjectSetUpdated method is called to signify to the collision system that this matrix has been updated (either explicitly by the application or via an animation update). We also assign the current world space transformation of this frame (calculated above) to the LastMatrix and CollisionMatrix members. We are saying that the current position of the dynamic object in the world (described by the passed world matrix) will also be the previous position when the collision update it first called. In other words, the object has not moved yet.

We do not want to assign LastMatrix an arbitrary position even if it will be overwritten the first time ObjectSetUpdated is called; we should set it to the current position of the parent frame. If we did not do that, then for the first update we might have large values in our velocity matrix calculated between the last matrix and the current matrix which could really throw off our response system. Furthermore, our terrain collision system could end up testing thousands of triangles unnecessarily purely because the swept ellipsoid would span a great distance in its first update forcing the collision system to think that the object has moved a great distance between the last and current updates. You should remember from the textbook that it is actually the previous matrix of the dynamic object (cached in CollisionMatrix) that is used for intersection testing.

```
                // Store the matrices well need for
                pObject->pCurrentMatrix=
                                        &((D3DXFRAME_MATRIX*)pFrame)->mtxCombined;

                pObject->LastMatrix      = mtxFrame;
                pObject->CollisionMatrix = mtxFrame;
                pObject->ObjectSetIndex  = m_nLastObjectSet + 1;
                pObject->IsReference     = false;
```

As the m_nLastObjectSet will contain the last ID assigned to an actor/object that was registered, we can add one to this amount to generate the new group ID for every dynamic object created from this hierarchy. Remember, the value of m_nLastObjectSet is never altered or incremented in this function; we are assigning the same ID to every dynamic object we create from this actor. The ID of every object in this group will be one greater than the ID of the previous group that was added. We also set the IsReference member to false as this is not a reference (we will discuss adding references in a moment).

We will now add the model space vertices of the mesh to the dynamic object vectors using the AddBufferData member. Take note of the first two parameters where we are passing the dynamic object's geometry arrays and not the static scene geometry arrays as before. Also notice that as the final parameter we pass an identity matrix so that the model space vertices are not transformed into world space. We want them to be stored in the dynamic object in model space because (as we saw in the textbook), the geometry will be transformed into world space on the fly during the EllipsoidIntersectScene call.

```
                // Add to the dynamic objects database
                if ( !AddBufferData(      *pObject->pCollVertices,
                                          *pObject->pCollTriangles,
                                          pVertices,
                                          pIndices,
                                          pMesh->GetNumVertices(),
                                          pMesh->GetNumFaces(),
                                          nVertexStride,
                                          nIndexStride,
                                          mtxIdentity ) ) throw 0;
```

Now that our dynamic object structure contains all the data it needs, let us add it to the collision system's dynamic object array and unlock the vertex and index buffers of the D3DX mesh.

```
                // Store the dynamic object
                m_DynamicObjects.push_back( pObject );

                // Unlock resources
                pMesh->UnlockVertexBuffer();
                pVertices = NULL;
                pMesh->UnlockIndexBuffer();
                pIndices = NULL;

            } // End try block
```

If anything went wrong in the above code and an exception is thrown, the following catch code block will be triggered. It releases the dynamic object structure and the STL vectors we allocated to contain its geometry. We also unlock the vertex and index buffers.

```
            catch (...)
            {
                // Is there an object already?
                if ( pObject )
                {
                    if ( pObject->pCollVertices  ) delete pObject->pCollVertices;
                    if ( pObject->pCollTriangles ) delete pObject->pCollTriangles;
                    delete pObject;

                } // End if object created

                // Unlock resources
                if ( pVertices ) pMesh->UnlockVertexBuffer();
                if ( pIndices  ) pMesh->UnlockIndexBuffer();

                // Return fatal error
                return false;

            }
        }
    }
```

If we reach this point in the code then every mesh container that was attached to the current frame has had a dynamic object created and has had its model space geometry arrays created. We now traverse into

the sibling and child lists as before, causing a cascade effect that allows this function to recursively visit the entire hierarchy and create dynamic objects from any meshes found. Each mesh is assigned the same object set ID and as such, all the dynamic objects in the actor will belong to the same object group in the collision system.

```
    // Process children and siblings
    if ( pFrame->pFrameSibling )
    {
        if ( !AddDynamicFrameHierarchy( pFrame->pFrameSibling, mtxWorld ) )
            return false;

    } // End if there is a sibling


    if ( pFrame->pFrameFirstChild )
    {
        if ( !AddDynamicFrameHierarchy( pFrame->pFrameFirstChild, mtxFrame ) )
            return false;

    } // End if there is a child

    // Success!
    return true;
}
```

We have now covered not only how to add dynamic objects to the collision system, but have also discussed how to add entire hierarchies of dynamic objects. There is one more dynamic object registration function that we must cover which allows an actor to be registered with the collision system by referencing an actor that has previous been registered.

CCollision::AddActorReference

A referenced dynamic object shares its geometry data with the dynamic object from which it was instanced. When an actor is added to the system, multiple dynamic objects will be created. To reference an actor, we simply call the AddActorReference function passing in the object set index of the actor that was originally added. We want this function to make a copy of every dynamic object with a matching object set index. We do not need to traverse the frame hierarchy of the actor to do this. We can simply loop though the collision system's dynamic object array searching for all dynamic objects that currently exist which match the object set index passed into the function. For each one that is found, we will create a new dynamic object. However, unlike the normal creation of dynamic objects, we will not allocate these dynamic objects their own vertex and triangle vectors. Instead, we will assign their pointers to the geometry buffers of the original dynamic object we are referencing. This makes the system more memory efficient.

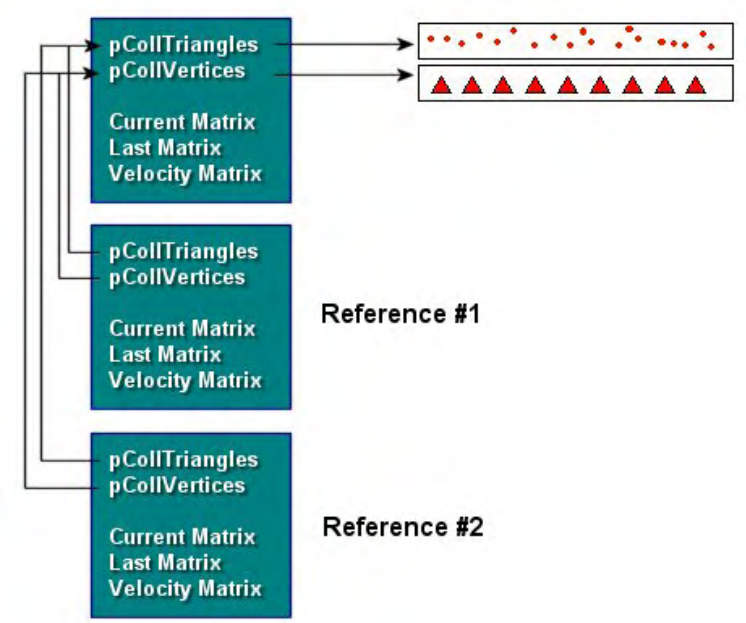


Figure 13.3

In Figure 13.3 we see how three dynamic objects might look in the collision system's dynamic object array. The topmost dynamic object was not created as a reference and therefore it has its own geometry buffers. The following two dynamic objects were registered using the AddActorReference function, passing in the object set index of the original dynamic object. As you can see, while these are dynamic objects in their own right (with their own object set index and matrix pointers), their geometry buffer pointers point at the buffers owned by the original dynamic object.

Although this makes sense, some confusion may be caused by the fact that the dynamic object's matrix pointer will be pointing to the same absolute frame matrix. The whole idea of references is to be able to place the same actor in the world in different locations and orientations without having to duplicate the geometry data. However, if each of our referenced dynamic objects points to the same world matrix (the absolute frame matrix in the actor's hierarchy) as the original dynamic object, would this not mean that all of our references will have to be in the same position in the world (negating the whole point of using references to begin with)?

While this is certainly a worthwhile observation to make, remember that the dynamic objects added to the system for the referenced actors will have different object set indices than the dynamic objects added to the system for the original actor. In the textbook we also discussed that when the application updates the position of an actor in any way (or applies an animation), it should immediately instruct the actor to update itself. This will cause the absolute frame matrices of the actor to be regenerated. If we have multiple CObjects using the same actor, we can see that this would cause the single actor to be updated many times in a given scene animation update, once for each CObject that stores a pointer to it. As we perform each update on the actor for each object that uses it (CScene::AnimateObjects), we also call the CCollision::ObjectSetUpdated function. This will update all the matrices of dynamic objects that were created from that actor or actor reference that was assigned that object set index. This function will use the matrix pointer to get the current position of the parent frame in the hierarchy which it will then use to build its LastMatrix, CollisionMatrix and VelocityMatrix members. The matrix pointer which points directly into the hierarchy is not needed by that group of dynamic objects any further in this update because its values have been cached. The frame matrix can therefore be updated by other objects in the scene after this point. The general pattern of actor updating is described below.

Imagine we have three CObjects in our scene and each one's actor pointer points to the same CActor. At the scene level, we know this actor is being referenced by three objects. That is, one set of geometry, and three instances of it. We would also want to register the same actor with the collision system as one real actor and two actor references. Assume that when we first register the actor with the collision

system we get back an object set index of 1. When we register the second actor as a reference, we get an object set index of 2. Finally, when we register the third actor (as our second reference) we get back an object set index of 3. If we also imagine that the original actor contained 10 meshes, we would now have 30 dynamic objects in our collision system. But there would only be 10 sets of geometry buffers (those from the original non-referenced actor).

We also know that each corresponding dynamic object from each of the three object groups in the collision system will store a pCurrentMatrix pointer pointing to the exact same frame matrix. However, as long as we update the position of each object separately, we will not have a problem. We just update the frames of that actor to reflect the pose of the object and then notify the collision system of a change so that it can grab a snapshot of the current values for each dynamic object's matrix at that point. The update strategy should be as follows:

1. For each CObject in CScene
    a. Apply animation to actor using this CObject's animation controller
    b. Set actor's world matrix to the CObject matrix and instruct actor to update its absolute matrices.
    c. Now that the actor is in its correct position for the reference, inform the collision system that it has been updated using CCollision::UpdateSetUpdated. We pass in the object set ID of this object reference. This function will extract the current state of each absolute matrix in the hierarchy that is being pointed at by each dynamic object and use it to generate its collision matrix, last matrix and its velocity matrix members. These will be used later in the current game loop to transform the dynamic object into world space for intersection testing.

As you can see, the fact that referenced and non-referenced dynamic objects with the same mesh point to the same physical frame matrix is not a problem, as long as we update each object group one at a time. This gives the CCollision::ObjectSetUpdated function a chance to generate its internal matrices based on a snapshot of the actor in the reference pose. From the collision system's perspective, it is almost as if we are posing the hierarchy in different poses for each object reference and recording the matrix information for each pose so that it can be used for intersection testing later.

Now that we know how the application deals with references both inside and outside the collision system, let us look at the code that allows us to add an actor reference to the collision system.

When an application calls this function it does not have to pass a pointer to an actor since this function has no need for the frame hierarchy. Every dynamic object that belongs to the original actor's object set will be duplicated in the collision system. Therefore, we just have to pass the object set index of the group we wish to reference and a world matrix describing where we would like the referenced actor to be placed in the scene. We must also pass the original world matrix that was used when we added the original actor (non-referenced) to the collision database. We will discuss why this is needed in a moment. Let us look at the first snippet of code.

```
long CCollision::AddActorReference( long ObjectSet,
                                    const D3DXMATRIX& mtxOriginalWorld,
                                    const D3DXMATRIX& mtxWorld )
{
```

```
    D3DXMATRIX      mtxInv;
    bool            bAddedData = false;

    // Generate the inverse of the original world matrix
    D3DXMatrixInverse( &mtxInv, NULL, &mtxOriginalWorld );
```

As you can see, the parameters to the function from left to right are the object set index of the group we wish to reference, the original world matrix that was used when the original group of objects was added to the collision system (the actor that was not added as a reference), and the world matrix for this reference.

Why do we need the world matrix of the original actor? In short, because we need to invert it so that we can undo the current world transform that is applied to the objects we are referencing. We will discuss this further in just a moment.

The next section of code sets up a loop to iterate through every dynamic object currently contained in the collision system's dynamic object array. It then compares the object set index assigned to each dynamic object in the array to see if it matches the object set index passed into the function. If there is no match we simply skip the dynamic object as it does not belong to the group we wish to reference. For each object that we do find that has an object set index that matches the one passed into the function, we know that it is one that we wish to reference, so we create a new dynamic object structure.

```
    // Iterate through our object database
    DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
    for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
    {
        DynamicObject * pObject = *ObjIterator;
        if ( !pObject ) continue;

        // We only reference if the set indices match
        if ( pObject->ObjectSetIndex == ObjectSet )
        {
            DynamicObject * pNewObject = NULL;

            // Catch Exceptions
            try
            {
                // Allocate a new dynamic object instance
                pNewObject = new DynamicObject;
                if ( !pNewObject ) throw 0;

                // Clear the structure
                ZeroMemory( pNewObject, sizeof(DynamicObject) );
```

Unlike non-referenced dynamic objects, we do not allocate the new object its own vertex and triangle arrays. Instead, we assign its pCollVertices and pCollTriangles members to point at the geometry buffers of the object being referenced. We also set the new objects IsReference Boolean member to true.

```
                // Setup referenced data
                pNewObject->IsReference    = true;
                pNewObject->pCollTriangles = pObject->pCollTriangles;
```

```
                pNewObject->pCollVertices  = pObject->pCollVertices;
                pNewObject->pCurrentMatrix = pObject->pCurrentMatrix;

                // Store the new object set index
                pNewObject->ObjectSetIndex = m_nLastObjectSet + 1;
```

Just as in the non-referenced case, we assign the new object's matrix pointer to point to the same matrix as the dynamic object it is instancing. That is, in the case of an actor, both the dynamic object and the referenced dynamic object point to the same absolute matrix stored in the mesh's owner frame. As just discussed, this does not cause a problem because the ObjectSetUpdated function will be called when each object is individually updated, allowing the collision system to take a snapshot of the frame matrix in the correct pose for that instance.

At the bottom of the above code, we also assign the new object a new object set ID which is NOT the same as the dynamic object we are referencing. We calculate this object set ID just as before; by adding one to the ID last issued by the collision registration functions. Note that if we are referencing an actor that has multiple meshes in its hierarchy, multiple dynamic object references will be created in the collision system, but each will have the same object set ID and belong to the same group, as we would expect. After all, when an object is updated in the collision system (CCollision::ObjectSetUpdated), we wish the matrix data for each dynamic object that belongs to that group to be re-calculated.

The next step is where we use that inverted matrix we calculated earlier and stored in the mtxInv local variable. This matrix contains the inverted world space transform of the original dynamic object we are referencing (the original actor). More accurately, it is the inverted world space transformation of the root frame of the actor we are referencing. So why do we need it?

As discussed in the previous function, we want to assign the LastMatrix and CollisionMatrix members of the dynamic object to a value that has legitimate meaning when we first call our collision update function. However, we cannot really set these members to the object's previous position as it has none; it is only just being created now. We also do not want to give these matrices arbitrary values because they will be used to create the velocity matrix of the dynamic object when the ObjectSetUpdated function is first called for this object group. If we assign these matrices a position in the world that is nowhere near the current position of the object in the first collision update, we will get a huge velocity vector which will wreak havoc in our response code. Furthermore, we do not want our terrain collision system thinking that the object has moved in the first update across a huge expanse of terrain since this would mean hundreds, perhaps thousands, of triangles would need to be temporarily built and tested for intersection.

So it makes sense to assign to the dynamic object's LastMatrix and CollisionMatrix members the current transform of the frame in the hierarchy which owns this mesh. However, unlike the AddDynamicFrameHierarchy function, which traversed the hierarchy and always had access to the absolute frame matrix, in this function we are not traversing the hierarchy and we are not combining matrices. Therefore, we do not know the world space position of the dynamic objects. That is, we do not know what the absolute matrix should contain in the reference pose. We are simply looping through the dynamic objects of the collision system. Although this function was passed a world matrix for this reference, it only describes the world matrix of the root frame in the reference position. Of course, the mesh in the actor will likely be offset from the root frame by several levels of transforms. What we need

to know is not the world matrix of the root frame of the actor in the reference pose (the mtxWorld parameter), but the world matrix of the owner frame in the reference position.

Although we do not have this information at hand, we do know the current world space position of the dynamic object we are *referencing* since this information is stored in its LastMatrix parameter. We also have the original world matrix that was assigned to that actor's root frame when it was registered with the collision system since we have passed it as the second parameter to this function (mtxOriginalMatrix). Finally, we have the inverse of this matrix which undoes the transformation that was applied to the root frame of the non-referenced actor when it was registered. Therefore, if we multiply this inverse matrix with the non-referenced dynamic object's LastMatrix pointer (the dynamic object we are copying), we are essentially subtracting the position and rotation of its root frame from the world space position of the non-referenced dynamic object. This transforms the actor and all its meshes into model space (the reference pose). In other words, we are left with a matrix for that object that describes its position and orientation relative to <0,0,0> in actor space.

```
                // Transform the matrices to ensure it starts in the correct place
                D3DXMatrixMultiply( &pNewObject->LastMatrix,
                                    &pObject->LastMatrix,
                                    &mtxInv );
```

Now that we have undone the transformation that was originally applied to the non-referenced object, we can transform it back out into world space using the world space matrix of the referenced object. The result is the current world space position of the corresponding frame in the hierarchy, positioned by the reference matrix (instead of the original matrix used to position the non-referenced actor).

```
                D3DXMatrixMultiply( &pNewObject->LastMatrix,
                                    &pNewObject->LastMatrix,
                                    &mtxWorld );
                pNewObject->CollisionMatrix = pNewObject->LastMatrix;
```

At this point, both the CollisionMatrix and the LastMatrix members of the new dynamic object store the current world space position of the dynamic object using the reference's own world transform.

With our dynamic object reference now created, we finally add it to the collision system's dynamic object array. If an exception was thrown in the above code, the catch block simply deletes the dynamic object structure.

```
                // Store the dynamic object
                m_DynamicObjects.push_back( pNewObject );

                // We've added, ensure we don't release
                pNewObject = NULL;

                // We successfully added some data
                bAddedData = true;

            } // End Try Block
```

```
            catch (...)
            {
                // Release new memory
                if ( pNewObject ) delete pNewObject;

                // Return failure.
                return -1;

            } // End Catch Block

        } // End if from matching set

    } // Next object
```

At this point, if the local Boolean variable bAddedData is set to true, we know that at least one of the currently existing dynamic objects was referenced and we have added a new object group. Therefore, we increment the m_nLastObjectSet and return the new group index to the caller. Otherwise, we return -1 indicating that the requested object group could not be referenced because it seemed not to exist.

```
    // Did we find anything to reference?
    if ( bAddedData )
    {
        // We have used another object set index.
        m_nLastObjectSet++;

        // Return the object index
        return m_nLastObjectSet;

    } // End if added references
    else
    {
        return -1;

    } // End if nothing added
}
```

Thankfully, we have covered all the geometry registration functions of the CCollision class and you should have a good understanding of where the static and dynamic objects live in the collision system and how they are accessed.

CCollision::Optimize

The optimize method can optionally be called after your application has registered all static and dynamic objects with the collision system. It simply compacts the STL vectors used by both the static database and each dynamic object to their actual size, eliminating wasted space introduced during registration.

This function will test the current size and the capacity of each STL vector used by the system. This includes the two vectors that contain the static vertices and triangles and the vectors of each dynamic object used to contain its geometry. Since the size of the vector describes how many elements have been added to it, and the capacity of the vector describes how many elements can be added to it, we just have

to change the capacity of the vector to equal to its current size. This is done using the vector::reserve method which removes the wasted space from the end of each vector.

```
bool CCollision::Optimize( )
{
    // For now, we simply remove any slack from our vectors, but you could
    // (for instance) weld all the vertices here too.
    try
    {
        // Remove array slack
        m_CollTriangles.reserve( m_CollTriangles.size() );
        m_CollVertices.reserve( m_CollVertices.size() );

        // Iterate through our object database
        DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
        for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
        {
            DynamicObject * pObject = *ObjIterator;
            if ( !pObject ) continue;

            // If there is a database, remove slack
            if ( pObject->pCollTriangles )
            pObject->pCollTriangles->reserve(pObject->pCollTriangles->size());

            if ( pObject->pCollVertices  )
            pObject->pCollVertices->reserve( pObject->pCollVertices->size() );

        } // Next object

    } // End Try Block

    catch ( ... )
    {
        // Simply bail
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

While this function is not vital to proper system functioning, it is recommended that your application call it after geometry registration to keep memory requirements as minimal as possible.

Updating Collision Querying with a CTerrain Handler

In the accompanying textbook we discovered that the CCollision::EllipsoidIntersectScene function is the heart of the collision detection process. It uses the EllipsoidIntersectBuffers method to perform collision determination on both the static geometry buffers and the geometry buffers of each dynamic object registered with the system. The code was already discussed in great detail, so we will not spend a great deal of time on it (or its helper functions) in this section. However, we will discuss the special case code we will add to allow us to perform intersection tests against any CTerrain objects that have been registered with the system.

We will first look at the updates to the EllipsoidIntersectScene function where we will discover that an additional collision phase has been added. In our textbook discussion, this function performed intersection testing in two phases. It would first loop through each dynamic object and perform intersection tests on each object's geometry buffers. Then it would perform intersection testing against the collision system's static geometry buffers. We will now add a third step (which will actually turn out to be performed as the first step in our revised implementation) which will process collisions against CTerrain objects. When we examined the CCollision::AddTerrain method earlier, we saw that it simply added the passed CTerrain pointer to an internal vector of terrain object pointers. No geometry from the terrain was added to the static or dynamic geometry buffers.

While adding terrain geometry to the collision system could be done simply by registering the terrain geometry with the static database just like we do with any other static geometry (using AddIndexedPrimitive), this is obviously not the most memory efficient design. Terrains are usually quite vast and are often comprised of many thousands of triangles. As programmers, we usually have a hard enough time as it is fitting such huge terrains in memory for the purposes of rendering. If we were to store a copy of each terrain triangle in the collision database, we would effectively double the memory overhead of using that terrain. Instead, we will use a procedural geometry approach at runtime that will make using terrains based on height map data very efficient. For terrains that are not built from height map data (e.g., static meshes designed in 3D Studio MAX™) these can still be registered with the collision system using the AddIndexedPrimitive methods.

For height map based terrains which can be loaded into our CTerrain class, we will store only the CTerrain pointer in the collision system. The EllipsoidIntersectScene function will now use a new function we will implement called CollectTerrainData to temporarily build the triangles of the terrain that need to be tested for intersection. This determination will be based on the position of the ellipsoid, its velocity vector, and the data contained in the height map. The ellipsoid and its velocity vector will be mapped into the image space of the terrain's height map and used to build a bounding box describing a region on the height map that contains potentially colliding triangles. In a given update, where the ellipsoid will have moved a very small distance from its previous position, this bounding box will span only a very small area of pixels in the height map. The bounded rectangular area of pixels can then be used to build quads for the terrain that falls within that box. This is exactly the same way we build the renderable terrain data from a height map. The only difference is that instead of building quads for every pixel in the height map, we are only using a very small subsection to build a temporary mini-terrain for the area we are interested in. Once this temporary buffer has been tested for collision, the data can then be released.

The CollectTerrainData function will return the vertices and triangles for a given sub-terrain in two buffers (STL vectors) that the EllipsoidIntersectScene function can then pass into the EllipsoidIntersectBuffers function for normal intersection testing. At this point, we will not concern ourselves with how the CollectTerrainData function builds its terrain data. First we will concentrate on the additions to the EllipsoidIntersectScene function. Although the entire function is shown below, we will only discuss the new code that has been added (shown in bold). The rest of the code has been discussed in detail in the accompanying textbook.

Collision::EllipsoidIntersectScene (Version 3)

In this first section of code we see two new lines that declare two local STL vectors. One will be used to hold the vertex data returned from the CollectTerrainData function and the other used to contain the triangle data. Note that these vectors are of the same type used for the static geometry in the collision system and the model space geometry of each dynamic object. We then calculate the inverse radius vector of the ellipsoid so that we can scale the vectors in and out of eSpace as required.

```
bool CCollision::EllipsoidIntersectScene( const D3DXVECTOR3 &Center,
                                          const D3DXVECTOR3& Radius,
                                          const D3DXVECTOR3& Velocity,
                                          CollIntersect Intersections[],
                                          ULONG & IntersectionCount,
                                          bool bInputEllipsoidSpace /* = false */,
                                          bool bReturnEllipsoidSpace /*= false*/ )
{
    D3DXVECTOR3 eCenter, eVelocity, eAdjust, vecEndPoint, InvRadius;
    float       eInterval;
    ULONG       i;

    // Vectors for terrain building
    CollVertVector VertBuffer;
    CollTriVector  TriBuffer;

    // Calculate the reciprocal radius
    InvRadius = D3DXVECTOR3( 1.0f / Radius.x, 1.0f / Radius.y, 1.0f / Radius.z );
```

In the next section of code we copy the passed ellipsoid position and velocity vectors into the local variables eCenter and eVelocity. If the bInputEllipsoidSpace Boolean parameter was set false it means that we would like this function to convert them into eSpace for us. In our code, the CollideEllipsoid function passes true for this parameter because it inputs both the ellipsoid position and velocity vector already in eSpace and therefore, the data is copied straight into the local variables.

```
    // Convert the values specified into ellipsoid space if required
    if ( !bInputEllipsoidSpace )
    {
        eCenter   = Vec3VecScale( Center, InvRadius );
        eVelocity = Vec3VecScale( Velocity, InvRadius );

    } // End if the input values were not in ellipsoid space
    else
    {
```

```
        eCenter   = Center;
        eVelocity = Velocity;

    } // End if the input values are already in ellipsoid space
```

Next set the initial intersection interval along the ray (the *t* value) to 1.0, meaning that the closest intersection is at the end of the velocity vector. If this is not modified to a smaller value by the intersection routines, the path of the ellipsoid along its desired velocity vector is free from obstruction and can be moved to its desired position. We also set the initial value of IntersectionCount to zero as we have not yet found any colliding triangles.

```
    // Reset ellipsoid space interval to maximum
    eInterval = 1.0f;

    // Reset initial intersection count to 0 to save the caller having to do this.
    IntersectionCount = 0;
```

Now we enter step one of the three detection steps. This is the new step that performs intersection testing against any CTerrain objects that have been registered with the collision system. This small section of code is virtually all the new code that has been added to this function. This is due to the fact that most of the new code is wrapped up in the CollectTerrainData method, which we will discuss in a moment.

We will first loop through every CTerrain object that has been registered with the collision system via the AddTerrain method. The pointers of each CTerrain object will be stored in the m_TerrainObjects member variable. This is an STL vector of type TerrainVector. In each iteration of the loop, we will extract the current CTerrain object being processed into the local pTerrain pointer for ease of access.

```
    // Iterate through our terrain database
    TerrainVector::iterator TerrainIterator = m_TerrainObjects.begin();
    for ( ; TerrainIterator != m_TerrainObjects.end(); ++TerrainIterator )
    {
        const CTerrain * pTerrain = *TerrainIterator;
```

We now have a pointer to the CTerrain object we want to test for intersections with our ellipsoid. The CollectTerrainData function will now be called to build and return the triangle data for the region of interest in the height map. The CollectTerrainData function must be passed the world space ellipsoid center position and velocity vector in order to do this. It uses these values to calculate the start and end positions of the ellipsoid in order to construct an image space bounding box on the height map. Because we currently have our ellipsoid position and velocity vector in eSpace, we must temporarily multiply them by the radius vector of the ellipsoid to put them back into world space. We then call the CollectTerrainData function, passing in the terrain object itself, the world space position and velocity vector, and the ellipsoid radius vector. As the final two parameters we pass the two local geometry buffers we allocated at the top of the function. When the function returns, these vectors will contain the triangle and vertex data needed to testing.

```
        // Get world space values
        D3DXVECTOR3 vecCenter   = Vec3VecScale( eCenter, Radius );
```

```
        D3DXVECTOR3 vecVelocity = Vec3VecScale( eVelocity, Radius );

        // Collect the terrain triangle data
        if ( !CollectTerrainData(   *pTerrain,
                                    vecCenter,
                                    Radius,
                                    vecVelocity,
                                    VertBuffer,
                                    TriBuffer ) ) continue;
```

If the function returns false, it means that the ellipsoid is not intersecting the overall terrain object and therefore no data could be collected from the height map. If the ellipsoid is not currently over the terrain, then mapping the bounding box of its start and end positions to image space would place it outside the entire height map. If the function returns false the ellipsoid could not possibly be colliding with this terrain and we just continue on to the next iteration of the loop and test any other terrain objects that may have been registered with the system.

If the CollectTerrainData function returns true, it means that some geometry was compiled from the terrain height map and should be tested for intersection. Luckily, we already have a function that performs all the intersection testing and interval recording -- the EllipsoidIntersectBuffers function. We can use it here as well to perform the tests against the temporary terrain buffers that were just built by CollectTerrainData.

```
        // Perform the ellipsoid intersect test against this set of terrain data
        EllipsoidIntersectBuffers(  VertBuffer,
                                    TriBuffer,
                                    eCenter,
                                    Radius,
                                    InvRadius,
                                    eVelocity,
                                    eInterval,
                                    Intersections,
                                    IntersectionCount );

        // Clear buffers for next terrain
        VertBuffer.clear();
        TriBuffer.clear();

    } // Next Terrain
```

The first and second parameters passed are the local temporary terrain geometry buffers that were just generated for this terrain object. This function will test every triangle in those buffers and record the closest colliding triangle's *t* value in the eIntersect variable. It will also store the triangle intersection information for this interval in the Intersections array.

After the EllipsoidIntersectBuffers function returns, we no longer need the temporary terrain geometry buffers, so we empty all the data they contain. They have served their purpose at this point and if any collision with the geometry did occur, the collision information will have been recorded in the Intersection array. Emptying the buffers lets us reuse them for any other terrain objects which need processing in future iterations of this loop.

At this point we have tested all the terrain objects and step one is complete. We now move on to step two where we test each dynamic object. This was all covered earlier.

```
    // Iterate through our triangle database
    DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
    for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
    {
        DynamicObject * pObject = *ObjIterator;

        // Calculate our adjustment vector in world space.
        vecEndPoint = (Vec3VecScale(eCenter, Radius)
                       + Vec3VecScale( eVelocity, Radius ));

        // Transform the end point
        D3DXVec3TransformCoord(&eAdjust, &vecEndPoint, &pObject->VelocityMatrix);

        // Translate back so we have the difference
        eAdjust -= vecEndPoint;

        // Scale back into ellipsoid space
        eAdjust  = Vec3VecScale( eAdjust, InvRadius );
```

We now have the adjustment vector which can be added to the velocity vector to compensate for the movement of the dynamic object between collision updates. So we call the EllipsoidIntersectBuffers function to test our ellipsoid against the geometry of the dynamic object. Notice how we extend the velocity ray by the adjustment vector while passing the parameter and that the first and second parameters are the model space buffers of the dynamic object. The final parameter is the world matrix of the dynamic object in its previous position. This technique is described in the textbook.

```
        // Perform the ellipsoid intersect test against this object
        ULONG StartIntersection
               = EllipsoidIntersectBuffers(     *pObject->pCollVertices,
                                                *pObject->pCollTriangles,
                                                eCenter,
                                                Radius,
                                                InvRadius,
                                                eVelocity - eAdjust,
                                                eInterval,
                                                Intersections,
                                                IntersectionCount,
                                                &pObject->CollisionMatrix );
```

We now loop through the intersection information that was added to the array for this dynamic object in the previous function call and adjust the new eSpace position and intersection points by the adjustment vector. This gives us the new position of the ellipsoid after it has been shunted back by any dynamic object that might have collided with it. Remember, the original collision test was done using a velocity vector that was extended by the opposite amount the dynamic object has moved from its previous position.

```
        // Loop through the intersections returned
        for ( i = StartIntersection; i < IntersectionCount; ++i )
        {
            // Move us to the correct point (including the objects velocity)
            // if we were not embedded.
            if ( Intersections[i].Interval > 0 )
            {
                // Translate back
                Intersections[i].NewCenter      += eAdjust;
                Intersections[i].IntersectPoint += eAdjust;

            } // End if not embedded

            // Store object
            Intersections[i].pObject = pObject;

        } // Next Intersection

    } // Next Dynamic Object
```

At this point, step two is complete and we have performed intersection tests against all terrain objects and all dynamic objects. Step three is the simplest -- we simply call EllipsoidIntersectBuffers one more time to perform intersection tests against our static geometry buffers.

```
    // Perform the ellipsoid intersect test against our static scene
    EllipsoidIntersectBuffers(      m_CollVertices,
                                    m_CollTriangles,
                                    eCenter,
                                    Radius,
                                    InvRadius,
                                    eVelocity,
                                    eInterval,
                                    Intersections,
                                    IntersectionCount );
```

We now have a CollIntersect array containing the information of all triangles that collided simultaneously at the smallest interval (eInterval). The new ellipsoid position and collision normals stored in this structure are currently in eSpace. If the caller passed false as the bReturnEllipsoidSpace boolean parameter, then it means they would like all the information stored in this array to be returned as world space vectors. When this is the case, we simply loop though each intersection structure in the compiled array and use the radius vector of the ellipsoid to scale the values from eSpace into world space. Our CollideEllipsoid function passes true as this parameter as it wants the information returned in eSpace. Thus, this code is never utilized in Lab Project 13.1.

```
    // If we were requested to return the values in normal space
    // then we must take the values back out of ellipsoid space here
    if ( !bReturnEllipsoidSpace )
    {
        // For each intersection found
        for ( i = 0; i < IntersectionCount; ++i )
        {

            // Transform the new center position and intersection point
```

```
            Intersections[ i ].NewCenter
                              =Vec3VecScale(Intersections[i].NewCenter,
                                            Radius );


            Intersections[ i ].IntersectPoint
                              = Vec3VecScale(   Intersections[i].IntersectPoint,
                                                Radius );

            // Transform the normal
            D3DXVECTOR3 Normal
                              = Vec3VecScale(   Intersections[i].IntersectNormal,
                                                InvRadius );
            D3DXVec3Normalize( &Normal, &Normal );

            // Store the transformed normal
            Intersections[ i ].IntersectNormal = Normal;

        } // Next Intersection

    } // End if !bReturnEllipsoidSpace

    // Return hit.
    return (IntersectionCount > 0);
}
```

At the very bottom of the function we return true if the number of intersections found is greater than zero; otherwise we return false. How the CollectTerrainData function works is the final piece of the puzzle and will be discussed next.

CCollision::CollectTerrainData

The CollectTerrainData function is tasked with finding the region of the passed CTerrain object that falls within a bounding box described by the movement of the ellipsoid. It will then build the terrain data for this region and add the vertex and triangle data to the passed buffers (VertBuffer and TriBuffer). As we have seen in the previous discussion, these buffers are then returned to the EllipsoidIntersectScene function where they are tested for intersection before being discarded. Basically what we are after is a cheap and simple way to reject most of the triangles from consideration so we only have to build a very small amount of temporary triangle data.

Since upgrading our CTerrain class in the previous chapter, a CTerrain object can now have a world matrix that positions and orients it in the world. We will transform the ellipsoid (sort of) into terrain space and compile a terrain space bounding box that describes a region of potential colliding triangles. We know that the terrain object also has a scale vector that describes the scale of the terrain geometry in relation to the height map used to create it. For example, a scale vector of (1,1,1) would mean that neighboring pixels in the image map would represent a space of 1 world space unit. A scale vector of (10,7,12) would mean that each group of four pixels represent a quad in terrain space that is 10 units wide (x axis) and 12 units deep (z axis). The values stored in the height map for each vertex would also be scaled by 7 to produce its terrain space height. Notice that we are referring to the scaled image data as describing the terrain space dimensions of the terrain and not the world space dimensions. This is

because the world matrix is also used to potentially rotate and translate the terrain geometry to position it in the world. Therefore, a quad that is 10x10 in terrain local space may be rotated about the world Y axis by 45 degrees. That is why we must transform the start and end positions of the ellipsoid into terrain space first so that we are working in the model space of the terrain. In this space, the quads of the terrains are aligned with the X and Z axes of the local coordinate system. Figure 13.4 shows how the center and movement vectors of the ellipsoid will be used to generate a terrain space bounding box for a (typically very small) region of the overall terrain.

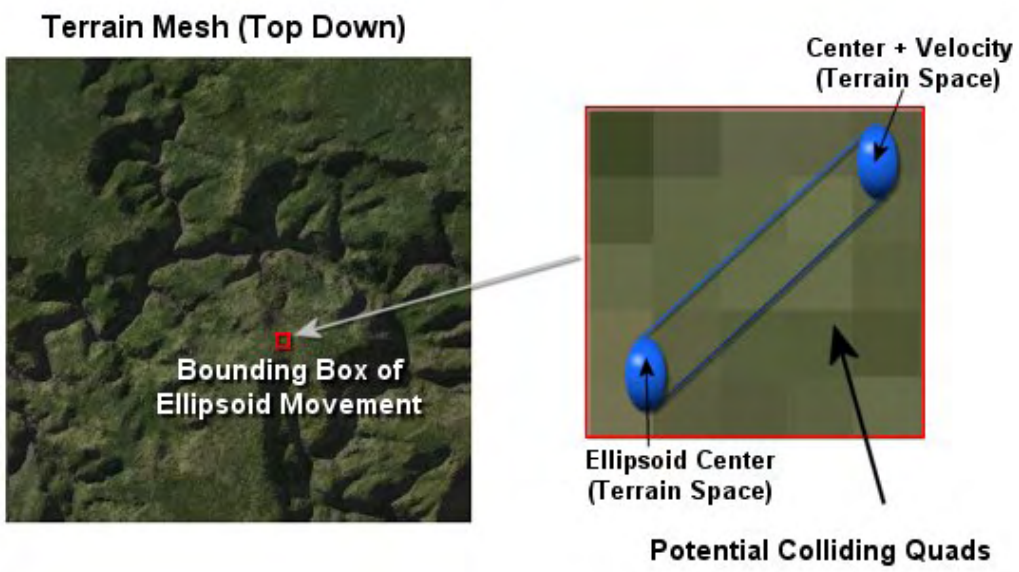


Figure 13.4

Since we know that the terrain's scale vector transforms the pixel positions in the height map into terrain space, dividing the dimensions of the terrain space bounding box by this scale vector will provide us with a bounding box in height map image space. Once we have this rectangle on the height map, we can simply loop through each row of contained pixels and build the triangle data in the exact same way we built the original terrain rendering geometry.

Let us now discuss this function a section at a time.

```
bool CCollision::CollectTerrainData(      const CTerrain& Terrain,
                                          const D3DXVECTOR3& Center,
                                          const D3DXVECTOR3& Radius,
                                          const D3DXVECTOR3& Velocity,
                                          CollVertVector& VertBuffer,
                                          CollTriVector& TriBuffer )
{
    D3DXMATRIX   mtxInverse;
    D3DXVECTOR3  tCenter, tVelocity, tvecMin, tvecMax, Vertex;
    long         nStartX, nEndX, nStartZ, nEndZ,
                 nX, nZ, nCounterX, nCounterZ, nPitch;
    float        fLargestExtent;

    // Retrieve the various pieces of information we need from the terrain
    const float *pHeightMap = Terrain.GetHeightMap();
```

```
    D3DXMATRIX   mtxWorld   = Terrain.GetWorldMatrix();
    D3DXVECTOR3  vecScale   = Terrain.GetScale();
    long         Width      = (long)Terrain.GetHeightMapWidth();
    long         Height     = (long)Terrain.GetHeightMapHeight();

    // Retrieve the inverse of the terrains matrix
    D3DXMatrixInverse( &mtxInverse, NULL, &mtxWorld );

    // Transform our sphere data, into terrain space
    D3DXVec3TransformCoord( &tCenter, &Center, &mtxInverse );
    D3DXVec3TransformNormal( &tVelocity, &Velocity, &mtxInverse );
```

In the first section of code (shown above) we retrieve all the information about the terrain we need; a pointer to its height map, the width and height of the height map, and the world matrix of the terrain object. We also fetch the terrain's scaling vector that describes the scaling that takes place to transform a pixel in the height map into a terrain space vertex position. The world space position and velocity vector of the ellipsoid have been passed into the function as parameters, but we need them in the terrain's local space. Remember that in world space, the terrain may be arbitrarily rotated or positioned by its world matrix, so we must make sure that the ellipsoid and the terrain are in the same space prior to compiling the bounding box. The obvious choice is terrain space because we can then easily transform the terrain space box into image space using the terrain's scale vector.

In order to do this, the ellipsoid center and velocity vectors will need to be multiplied by the terrain's inverse world matrix. Thus, in the above section of code we invert the terrain matrix and transform the vector into terrain space. The terrain space vectors are stored in local variables tCenter and tVelocity. If we look at Figure 13.4, we can see that tCenter represents the bottom left blue ellipsoid and tCenter + tVelocity describes the position of the top right blue ellipsoid.

Of course, we cannot just take the start and end points of the ellipsoid into account when compiling our terrain space bounding box. The vectors describe only the extents of the ellipsoid's center point as it travels along the terrain space velocity vector. As we know, an ellipsoid has a width, height and depth described by the radius vector that was also passed into the function. Therefore, if we could transform the ellipsoids radius vector into terrain space also, we would know that the extents of the bounding box along any of its three axes can be found by adding and subtracting the terrain space radius vector from the source and destination locations of the ellipsoid and recording the smallest and largest x, y and z values. This gives us the box shown in Figure 13.4 where it bounds the start and end center points and the radii of the ellipsoid surrounding those points.

So in order to compile our terrain space bounding box we must also transform the radius vector of the ellipsoid into terrain space. Then we have everything we need to start compiling our bounding box. You would be forgiven for thinking that we can transform the radius of the ellipsoid into terrain space simply by transforming the ellipsoid radius vector by the terrain's inverse matrix. Unfortunately, this is not the case, as transforming the ellipsoid radius vector in this way will produce a very different shaped ellipsoid in terrain space.

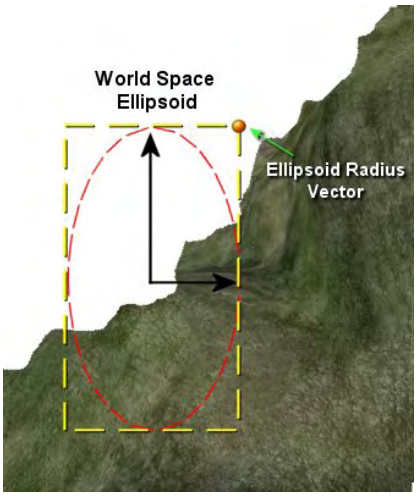


Figure 13.5

In Figure 13.5 we see both an ellipse and a terrain in world space. We will reduce the problem to 2D here for ease of explanation. The terrain's matrix has it rotated 45 degrees so that our ellipse is actually colliding into it at an angle. We know we can transform the center of the ellipsoid into terrain space using the inverse world matrix of the terrain but the ellipsoid radius is a completely different matter. If we think about the radius vector, we can see that while it is used to describe three radius values, if we think of it as a 3D vector, it actually describes a location at the tip of a bounding box that encases the ellipsoid. In Figure 13.5 the width radius is 1 and the height radius is 2 and therefore, the radius vector (1, 2) actually describes a location 1 unit along the X axis and 2 units along the Y axis. This is shown as the orange sphere at the top right corner of the bounding box. Inside the box we see the actual ellipsoid when the components of this vector are used to describe an ellipse radius.

In terrain space, the terrain will no longer be rotated; it will be perfectly aligned with the X, Y, and Z axes of the coordinate system and the ellipse shown in Figure 13.5 will be rotated forward 45 degrees. If we look at the two back arrows emanating from the center of the ellipse, we can see that they show its width and height axes. We might think that rotating the radius vector 45 degrees to the right would rotate these axes also thus providing us with a perfect terrain space bounding volume. However, this is not the case. As discussed, the world space radius vector describes only that orange sphere in Figure 13.5. When we rotate it, we are actually rotating the orange sphere by 45 degrees. In terrain space, this will now describe the top right extent of a bounding box that encloses the new terrain space ellipsoid. This is not remotely similar to the ellipsoid we were after as Figure 13.6 clearly shows.

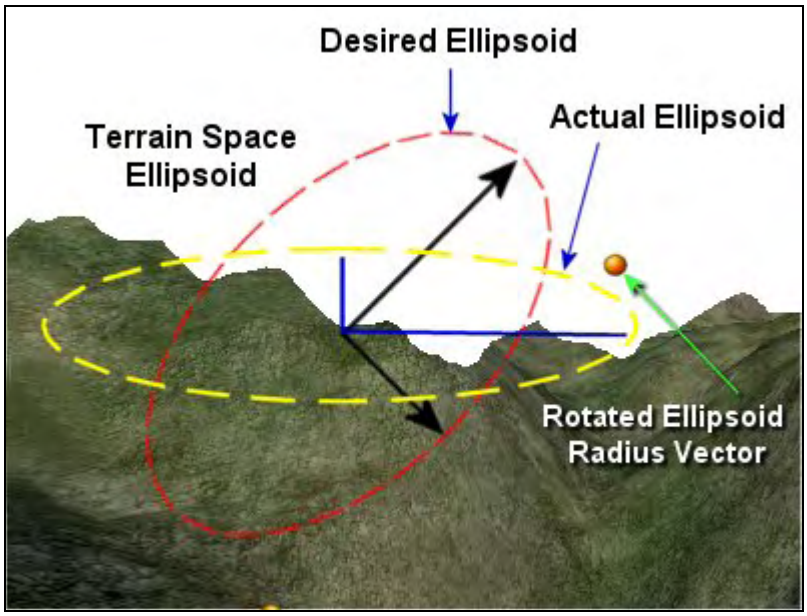


Figure 13.6

In Figure 13.6 the terrain space ellipse that we were actually hoping to get is shown as the red dashed ellipse complete with its rotated axis vectors. However, the orange sphere shows the radius vector plotted as a position describing the new top right extent of a bounding box that encases the new terrain space ellipse (post-rotation). Therefore, if we imagine this as being one corner of a box that surrounds the center of the ellipsoid, we can see the actual ellipse this describes as the yellow dashed ellipse. The ellipse has been severely squashed vertically and it is certainly not conservative. That is, the ellipsoid we ideally wanted does not fit inside the shape we actually get after rotation. Therefore, it is dangerous to use this approximation. We may reject polygons that do intersect the ellipsoid in world space but which fail to be added to the collision list because they do not intersect our terrain space ellipsoid.

The solution is simple, but comes with the cost of perhaps being a little too conservative. The problem here is the rotation of the terrain in world space and how to make sure that our ellipsoid in terrain space is at least big enough to force the compilation of a bounding box that will cause all polygons that could potentially collide with our ellipsoid to be added to the arrays. What we will do is simplify our problem by making our ellipsoid a sphere in terrain space. The rotation of the terrain in relation to a sphere is not significant as it has a radius equal in all directions. We need this sphere to be large enough to contain what our ellipsoid should like in terrain space when rotated at any angle. All we have to do then is simply take the largest component of the world space ellipsoid radius vector and essentially use a sphere that has a radius as large (see Figure 13.7).

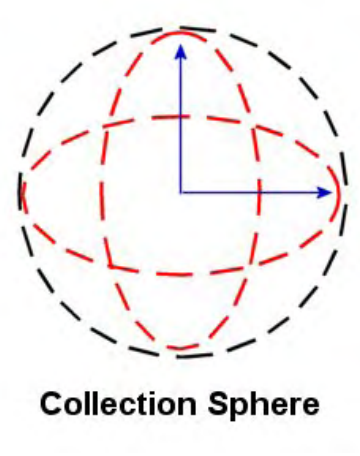


Figure 13.7

In this diagram we show two ellipsoids. The taller ellipsoid is the actual ellipsoid in world space and the wide ellipsoid shows the maximum width the ellipsoid could be in terrain space if the terrain were rotated 90 degrees. If we take the largest radius dimension of the world space ellipsoid vector and use this to build a sphere, we will have a sphere that completely encapsulates any rotation that may be applied to the original ellipsoid when transformed into terrain space.

Of course, we do not actually need to build a sphere as all we are after is the radius value. Once we have this, we can both add and subtract this value from the terrain space start and end positions of the center of the ellipsoid and record the maximum and minimum terrain space extents that we find.

The following code compiles the terrain space bounding box. It first tests each component of the world space ellipsoid radius vector to find which is the largest. This will be the radius of our hypothetical terrain space sphere.

```
    // Find the largest extent of our ellipsoid.
    fLargestExtent = Radius.x;
    if ( Radius.y > fLargestExtent ) fLargestExtent = Radius.y;
    if ( Radius.z > fLargestExtent ) fLargestExtent = Radius.z;
```

Now that we have the radius of our terrain space sphere, we will compile a bounding box by finding the minimum and maximum world space positions by adding and subtracting this radius vector from the terrain space start and end locations of the ellipsoid's center point. We start by first setting the minimum

and maximum vectors of this bounding box to values which describe an inside-out box that will be snapped to the correct size as soon as the first location tests are performed.

```
    // Reset the bounding box values
    tvecMin = D3DXVECTOR3( 9999999.0f, 9999999.0f, 9999999.0f );
    tvecMax = D3DXVECTOR3( -9999999.0f, -9999999.0f, -9999999.0f );
```

First we will add the radius of the sphere to the starting position of our ellipsoid. If this is larger than the current maximum we have recorded for that axis so far, we record the new extent. Note that this is done on a per-axis basis since we need the x, y, and z minimum and maximum extents to create a box.

```
    // Calculate the bounding box extents of where the ellipsoid currently
    // is, and the position it will be moving to.
    if ( tCenter.x + fLargestExtent > tvecMax.x )
            tvecMax.x = tCenter.x + fLargestExtent;

    if ( tCenter.y + fLargestExtent > tvecMax.y )
            tvecMax.y = tCenter.y + fLargestExtent;

    if ( tCenter.z + fLargestExtent > tvecMax.z )
            tvecMax.z = tCenter.z + fLargestExtent;
```

We next test to see if subtracting the radius of our sphere from the ellipsoid's starting position provides a new minimum extent for any of the axes.

```
    if ( tCenter.x - fLargestExtent < tvecMin.x )
            tvecMin.x = tCenter.x - fLargestExtent;

    if ( tCenter.y - fLargestExtent < tvecMin.y )
            tvecMin.y = tCenter.y - fLargestExtent;

     if ( tCenter.z - fLargestExtent < tvecMin.z )
            tvecMin.z = tCenter.z - fLargestExtent;
```

Now we test to see if adding the sphere radius to the destination location of the ellipsoid's center point (calculated as tCenter + tVelocity) provides a new maximum extent for any axis.

```
     if ( tCenter.x + tVelocity.x + fLargestExtent > tvecMax.x )
            tvecMax.x = tCenter.x + tVelocity.x + fLargestExtent;

     if ( tCenter.y + tVelocity.y + fLargestExtent > tvecMax.y )
            tvecMax.y = tCenter.y + tVelocity.y + fLargestExtent;

     if ( tCenter.z + tVelocity.z + fLargestExtent > tvecMax.z )
            tvecMax.z = tCenter.z + tVelocity.z + fLargestExtent;
```

And finally we test to see if subtracting the radius of our sphere from the destination location of the ellipsoid's center point provides us with a new minimum extent for any axis.

```
     if ( tCenter.x + tVelocity.x - fLargestExtent < tvecMin.x )
```

```
            tvecMin.x = tCenter.x + tVelocity.x - fLargestExtent;

    if ( tCenter.y + tVelocity.y - fLargestExtent < tvecMin.y )
            tvecMin.y = tCenter.y + tVelocity.y - fLargestExtent;

    if ( tCenter.z + tVelocity.z - fLargestExtent < tvecMin.z )
            tvecMin.z = tCenter.z + tVelocity.z - fLargestExtent;
```

With floating point inaccuracies being what they are, we would hate the above bounding box to miss a vertex later on just because it was outside the box by some very small epsilon value (0.000001 for example). Therefore, we will give ourselves a bit of padding by inflating the box 2 units along each axis; one unit for each axis in the positive direction and one for each axis in the negative direction.

```
    // Add Tolerance values
    tvecMin -= D3DXVECTOR3( 1.0f, 1.0f, 1.0f );
    tvecMax += D3DXVECTOR3( 1.0f, 1.0f, 1.0f );
```

We now have a bounding box in terrain space encompassing all the triangles that might intersect the movement of the ellipsoid. Any triangles outside this box cannot possibly collide, so sending them through our intersection routines would be unnecessary.

Next we need to transform our box into the image space of the height map. This is very easy to do. The terrain object's scale vector was used to transform a pixel in the height map into terrain space. All we had to do was multiply the x and y coordinate of the pixel by the scaling vector and we get the terrain space X and Z vertex coordinates generated from the pixel. The value stored in the pixel itself was also multiplied by the scale vector to create the height of the vertex (Y position) in terrain space. Thus, all we have to do to turn our terrain space 3D bounding box into a 2D rectangle on the height map, is reverse the process and divide the X and Z extents of this box by the scaling vector. We also snap the results to integer pixel locations on the height map as shown below.

```
    // Calculate the actual heightmap start and end points
    // (ensure we have enough surrounding points)
    nStartX = (long)(tvecMin.x / vecScale.x) - 1;
    nStartZ = (long)(tvecMin.z / vecScale.z) - 1;
    nEndX   = (long)(tvecMax.x / vecScale.x) + 1;
    nEndZ   = (long)(tvecMax.z / vecScale.z) + 1;
```

Notice how after generating the integer extents of the 2D box along both the X and Z axes, we subtract and add one to the minimum and maximum extents, respectively. This is to make sure that we always have at least four different corner points so that at least a single quad can be built. If we imagine for example that the ellipsoid is small and has no velocity, it is possible that the four extents of the terrain space bounding box, when transformed into image space and snapped to integer values, could all map to the same single pixel location in the height map. Remember that a pixel in the height map represents a vertex, so we need at least four unique points to build a quad. This addition and subtraction makes sure this is always the case.

In the next section of code we clamp the extents of the bounding box so that its extents are within the image of the height map. We certainly do not want to be trying to access pixel locations like x=600 if

the image is only 300 pixels wide. We also do not want to try and access pixel locations like x=-10 since there are no negative pixels in image space.

```
    // Clamp these values to prevent array overflow
    nStartX = max( 0, min( Width - 1, nStartX ) );
    nStartZ = max( 0, min( Height - 1, nStartZ ) );
    nEndX   = max( 0, min( Width - 1, nEndX ) );
    nEndZ   = max( 0, min( Height - 1, nEndZ ) );
```

At this point, we have four integer values which describe the corner points of our 2D bounding box where (nStartZ, nStartX) = Top Left and (nEndZ, nEndX) = Bottom Right. Remember at this point that nStartZ and nEndZ describe the image space Y axis, which increases top to bottom. Before we start stepping through these pixels and building vertices from each one, we should first make sure that the bounds of the box are not degenerate. That is, if the width or height of the box is zero then we just return true. We have no triangles to add from this terrain to the collision list.

```
    // Return if the bounds are degenerate (no op)
    if ( nEndX - nStartX <= 0 || nEndZ - nStartZ <= 0 ) return true;
```

Now it is time to start stepping through the rows and columns of pixels contained inside the rectangle on the image.

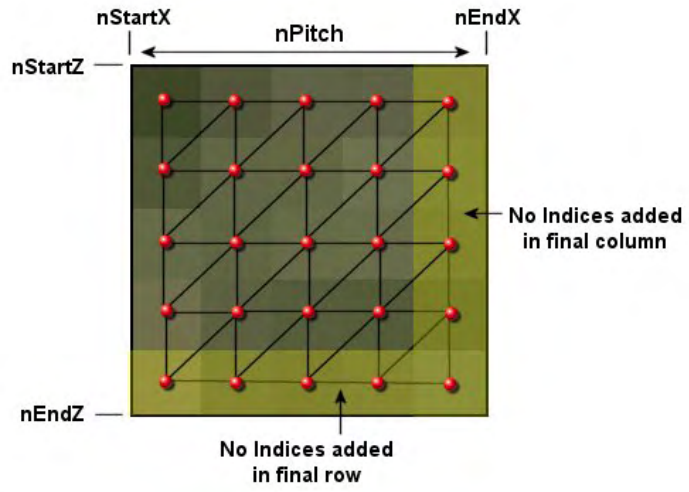


Figure 13.8

Figure 13.8 shows how we will build the vertex and triangle data for the vertices inside our box. Starting at the top left corner (nStartX , nStartY) we will work along each row and along each column. For each pixel in the height map we will add a vertex along with six indices describing the quad formed by the vertex, its right neighbor, the vertex below it and the vertex below and to the right of it. Since the top left vertex of any quad is the first to be added, when building the two triangles at the vertex, our triangles will be indexing into vertices that we have not yet added.

For example, look at the top left vertex in Figure 13.8, which shows the vertex that will be added in the first iteration of the loop. Not only will we create a vertex here, we will also create the entire top left quad (the group of four vertices in the top left corner). Although these other three vertices have not been added yet, we know where they will be positioned in the vertex buffer because we know how many vertices will be created in a given row (nPitch).

In Figure 13.8 we can see that the image space bounding box has dimensions of 5x5, so we will be adding 5 rows of 5 vertices. Therefore, if there are 5 vertices in each row, the pitch of our vertex array will be 5. When we visit the top left vertex, this will be vertex 0 in the array. Although the other three

vertices forming the quad have not yet been added (but will be added in future iterations of the loop) we do know the locations where each of these vertices will be in the vertex array and therefore we have the information with which to build the quad which uses the current vertex as the top left corner. We know for example that when processing a vertex at position *n*, the four vertex positions of all vertices forming the quad will eventually be in our vertex array at:

Top Left = n (Current Vertex)
Top Right = n+1
Bottom Left = n+pitch
Bottom Right = n+pitch+1

If we were currently processing the pixel at x=2 and y=10 in the rectangle, the vertices that would be added to the vertex buffer (assuming a pitch of 5 vertices per row) in the inner loop iteration would be:

Top Left = 2+10*pitch
Top Left = 52

In other words, the vertex we are currently processing will be added at location 52 in the vertex array. Furthermore, although they have not yet been added, the other three vertices comprising the quad will be added to the vertex array at positions:

Top Right = 2+1 + (10*pitch) = 53

Bottom Left = 2 + ((10+1)*pitch) = 67

Bottom Right = (5+1) + ((10+1)*pitch) = 68

As you can see, although we have only made it as far as processing vertex 52 (2nd vertex in the 10th row of our box), we can calculate the indices of the four vertices needed to comprise the quad. But this means that we must make sure we do not try to add any triangle data when processing the last vertex in each column and the last row of vertices. If you look at Figure 13.8 once again, you will see that the last column and the last row of vertices are highlighted yellow. When processing these vertices, we will not add any indices at all (only the vertices) because the quads that these vertices are a part of have already been added when processing the previous vertex in the column or row.

Let us now loop through each pixel in our bounding box and create the vertex and triangle data for it. First we calculate nPitch, which contains the number of vertices that will be in each row of the vertex array we will compile. We calculate this by adding 1 to the width of the image space bounding box.

```
    // Catch all exceptions
    try
    {
        // Store pitch value (to save us having to calculate each time
        nPitch = (nEndX - nStartX) + 1;
```

Notice how we add 1 to the result since this is the count of the number of vertices that will comprise each row of our vertex buffer, and we do not want it to be zero based. For example, if StartX=10 and

EndX=14 then this means there are actually 5 vertices on each row (the vertices at locations 10,11,12,13 and 14). If we were to just subtract the start dimension from the end dimensions we would get 14-10=4, which is not correct.

Now that we know the start and end positions of our bounding box along the X and Y axes of image space, we will loop through each row (outer loop) and each column (inner loop).

```
        // Build triangle data for each of the quads this falls into
        for ( nZ = nStartZ, nCounterZ = 0; nZ <= nEndZ; ++nZ, ++nCounterZ )
        {
            // For each column
            for ( nX = nStartX, nCounterX = 0; nX <= nEndX; ++nX, ++nCounterX )
            {
```

If we are not about to add the last vertex in a row or the last vertex in a column, we will allocate enough room in our triangle buffer to add two more CollTriangle structures. This is because we are about to create the quad for which the current vertex we are processing forms the top left corner. Notice how we allocate a temporary triangle structure that will be reused to add the data for each triangle of this quad to the triangle buffer.

```
                // Do not add triangle data for last column / row
                if ( nZ < nEndZ && nX < nEndX )
                {
                    CollTriangle Triangle;
                    ZeroMemory( &Triangle, sizeof(CollTriangle) );

                    // Grow the triangle buffer if required
                    if ( TriBuffer.capacity() < TriBuffer.size() + 2 )
                    {
                        // Reserve extra space
                        TriBuffer.reserve(TriBuffer.capacity()+m_nTriGrowCount );

                    } // End if should grow buffer
```

Each CollTriangle structure has a three element indices array, so we will add the indices for the first triangle. We will then add the triangle structure to the passed triangle buffer (STL vector).

```
                    // Build first triangle
                    Triangle.Indices[2] = nCounterX + nCounterZ * nPitch;
                    Triangle.Indices[1] = (nCounterX + 1) + nCounterZ * nPitch;
                    Triangle.Indices[0] = nCounterX + (nCounterZ + 1) * nPitch;

                    // Store first triangle
                    TriBuffer.push_back( Triangle );
```

Now we will reuse the CollTriangle structure to add the indices of the second triangle in the quad and add that to the triangle buffer also.

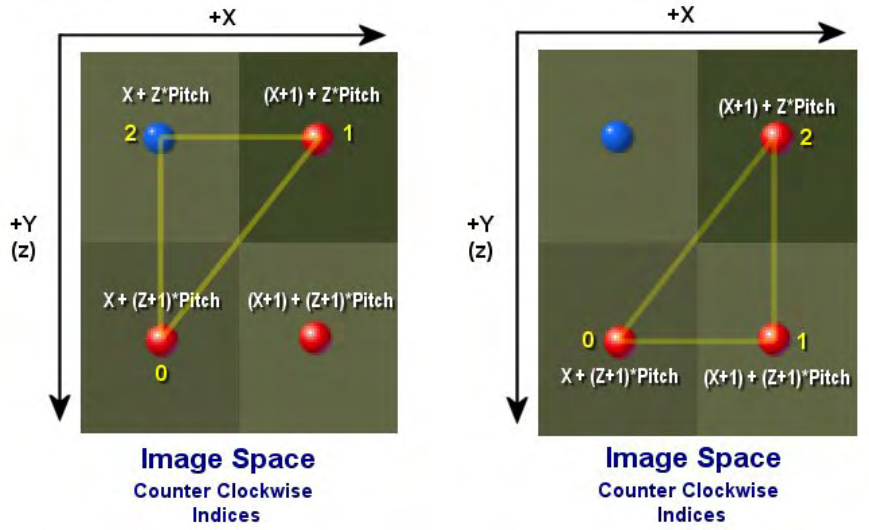
```
                    // Build second triangle
                    Triangle.Indices[2]= (nCounterX + 1) + nCounterZ * nPitch;
                    Triangle.Indices[1]= (nCounterX+1) + (nCounterZ + 1) * nPitch;
```

```
                    Triangle.Indices[0]=  nCounterX + (nCounterZ + 1) * nPitch;

                    // Store second triangle
                    TriBuffer.push_back( Triangle );

                } // End if last column / row
```

At this point we have added the two triangles that form the quad. Remember, this quad is only created and added if the current vertex we are processing (nCounterX, nCounterY) is not the final vertex in a row or the final vertex in the column (Figure 13.8).

You might have noticed that we are defining the indices of each triangle in a counter clockwise winding order. Figure 13.9 should help you visualize both the first and second triangle we add in the above code.

Figure 13.9



Triangle 1 **Triangle 2**

The vertices have been clearly indexed by each triangle in a counter clockwise order in image space. However, we must remember that the image space Y axis is equivalent to our terrain space Z axis. Furthermore, while the image space Y axis increases as it goes down the screen, in terrain space, if we were looking down on the terrain mesh from above, the Z axis would decrease as it headed down the screen. Therefore, as discussed back in 3D Graphics Module I, when we first examined terrain building using height maps, when the image is mapped to terrain space (when the image space Y axis is used as the terrain space Z axis) there is an implied 'flip' of the terrain about the image space Y axis. If we change the direction of the image space Y axis so that it is facing the opposite direction (as is the case when we build a 3D mesh from this image) you will see that the triangles are now defined with a clockwise winding order. Therefore, we define them counter clockwise in image space so that when the image is flipped during the transformation to terrain space, the winding order changes and all is well. Figure 13.10 shows the two images flipped along the image space Y axis. This is what happens when the image space Y coordinates are mapped to the terrain space Z axis.

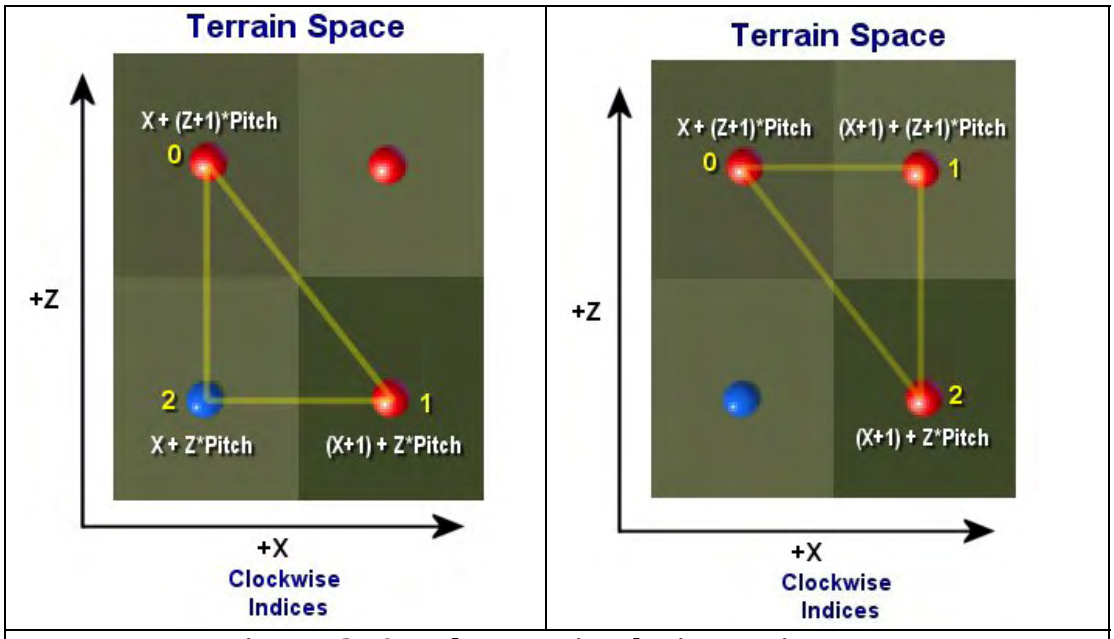


Figure 13.10 : The two triangles in terrain space

So we have added the quad (if applicable) whose top left corner is represented by the current pixel we are visiting. Now it is time to create and add the vertex itself. First we make sure there is enough room in the vertex array to add another vertex. If not, we resize the vector.

```
                // Grow the vertex buffer if required
                if ( VertBuffer.capacity() < VertBuffer.size() + 1 )
                {
                    // Reserve extra space
                    VertBuffer.reserve( VertBuffer.capacity()+m_nVertGrowCount );

                } // End if should grow buffer
```

The terrain space vertex position along the X and Z axes is simply the X and Y coordinate of the pixel scaled by the terrain object's scale vector. The Y coordinate of the vertex is the value of the pixel itself (stored in the pHeightMap array) scaled by the Y component of the terrain object's scale vector.

```
                // Calculate the vertex
                Vertex = D3DXVECTOR3( (float)nX * vecScale.x,
                                       pHeightMap[nX+nZ * Width] * vecScale.y,
                                      (float)nZ * vecScale.z  );
```

At this point, we have the terrain space vertex, but we need to return our triangle information in world space. So we multiply the vertex by the terrain object's world matrix before finally adding it to the vertex array.

```
                // Transform into world space
                D3DXVec3TransformCoord( &Vertex, &Vertex, &mtxWorld );
```

```
                // Add the vertex to the buffer
                VertBuffer.push_back( Vertex );

            } // Next Column

        } // Next Row
```

If we get to this point in the function we have successfully added a rectangular region of world space triangles to the passed vectors.

Our collision system also requires that each triangle have a normal which will be used as the slide plane normal in the case of impacts between the ellipsoid and the interior of triangle. Therefore, as a last step, we will loop through all the triangle structures we have just added to the triangle vector and generate a normal for each one. We do this using the same technique we have used many times before. That is, we use the vertices of the triangle to create two edge vectors that are tangent to the triangle surface and perform the cross product on them to generate a vector that is perpendicular to the surface. We then normalize the result.

```
        // Calculate resulting normals
        D3DXVECTOR3 v1, v2, v3, Edge1, Edge2;
        CollTriVector::iterator Iterator = TriBuffer.begin();
        for ( ; Iterator != TriBuffer.end(); ++Iterator )
        {
            // Get a REFERENCE to the underlying triangle
            CollTriangle & Triangle = *Iterator;

            // Retrieve vertices
            v1 = VertBuffer[ Triangle.Indices[0] ];
            v2 = VertBuffer[ Triangle.Indices[1] ];
            v3 = VertBuffer[ Triangle.Indices[2] ];

            // Calculate two edge vectors
            Edge1 = v2 - v1;
            Edge2 = v3 - v1;

            // Generate cross vector
            D3DXVec3Cross( &Triangle.Normal, &Edge1, &Edge2 );
            D3DXVec3Normalize( &Triangle.Normal, &Triangle.Normal );
        } // Next Triangle

    } // End Try Block

    catch ( ... )
    {
        // Return false. We failed.
        return false;

    } // End Catch

    // We added data successfully.
    return true;
}
```

At the end of the function you can see the catch block that simply returns false should an exception be thrown during execution.

Using the Collision System with Dynamic Objects

We have already discussed all the methods that allow our application to register dynamic object groups with the collision database. We discovered that when an object (or a hierarchy of objects) is registered with the collision system, an object set index is returned to the application. This index is the handle by which the application informs the collision system that it has altered the matrix (matrices) of an object (group of objects).

Some exterior entity (the application, a CObject, a D3DXFRAME, etc.) owns the matrix that contains the world space transformation for a dynamic object; the collision system simply maintains a pointer to it for access during dynamic object updates. The application can feel free to move the dynamic object about in the world by setting that matrix explicitly or by playing animations on the actor which owns the object.

CCollision::ObjectSetUpdated

Whenever the application updates the matrix of a dynamic object, it must inform the collision system by calling its ObjectSetUpdated method. The only parameter that need be passed is the object set index of the group that has had its matrix (matrices) updated. If the object set index represents a hierarchy of dynamic objects (an actor), then calling this function will cause the collision matrix of every effected dynamic object to be recalculated. Here is the code to the function.

```
void CCollision::ObjectSetUpdated( long Index )
{
    D3DXMATRIX mtxInv;

    // Update dynamic object matrices
    DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
    for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
    {
        DynamicObject * pObject = *ObjIterator;

        // Skip if this doesn't belong to the requested set.
        if ( pObject->ObjectSetIndex != Index ) continue;

        // Generate the inverse of the previous frames matrix
        D3DXMatrixInverse( &mtxInv, NULL, &pObject->LastMatrix );

        // Subtract the last matrix from the current to give us the difference
        D3DXMatrixMultiply(   &pObject->VelocityMatrix,
                              &mtxInv, pObject->pCurrentMatrix );

        // Store the collision space matrix
        pObject->CollisionMatrix = pObject->LastMatrix;
```

```
        // Update last matrix
        pObject->LastMatrix = *pObject->pCurrentMatrix;

    } // Next Object
}
```

It loops though every dynamic object in the collision system's dynamic object array. For each one it finds with a matching object set index, it calculates the new velocity matrix. As discussed in the textbook, the velocity matrix describes the relative transformation from its previous position/orientation. We calculate this by inverting its previous world matrix and multiplying it with the new updated world matrix. This provides a relative movement matrix describing how the object has moved between this update and the previous one. We store this data in the velocity matrix member of the dynamic object. As we saw in the EllipsoidIntersectScene function, this matrix is used to extend the swept sphere so that intersection tests can be performed against the geometry of the dynamic object in its previous position. We then store the current previous world matrix in the collision matrix member. The collision matrix describes the previous world transform of the object and will be used alongside the velocity matrix by the intersection routines. We then copy the new current world matrix into the last matrix member so that the process can be repeated again and again every time the object group is updated. Remember, only the collision matrix and the velocity matrix are used by the intersection routines. The last matrix and current matrix members are used only to generate these two matrices.

CCollision::SceneUpdated

Rather than force the application to call the ObjectSetUpdated method for each updated object, a convenience function has been added that allows the application to tell the collision system that all dynamic objects should have their matrix states updated simultaneously with a single function call.

The SceneUpdate method of the CCollision class (shown below) is almost identical to the previous function. The difference is that it is not passed an object set index. It just updates the matrices for every dynamic object registered with the system.

```
void CCollision::SceneUpdated( )
{
    D3DXMATRIX mtxInv;

    // Update dynamic object matrices
    DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
    for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
    {
        DynamicObject * pObject = *ObjIterator;

        // Generate the inverse of the previous frames matrix
        D3DXMatrixInverse( &mtxInv, NULL, &pObject->LastMatrix );

        // Subtract the last matrix from the current to give us the difference
        D3DXMatrixMultiply(   &pObject->VelocityMatrix,
                              &mtxInv,
                              pObject->pCurrentMatrix );
```

```
        // Store the collision space matrix
        pObject->CollisionMatrix = pObject->LastMatrix;

        // Update last matrix
        pObject->LastMatrix = *pObject->pCurrentMatrix;

    } // Next Object
}
```

Applications should ***never*** use this function with actors that have been registered as dynamic references. This is a simple convenience function that applied only in cases where actor references are not being used. Do you see why this must be the case?

To keep things simple, imagine a single mesh actor that has been initially registered with the collision system and then referenced 9 times (via AddActorReference). This means we will have 10 dynamic objects in the system. The first registered dynamic object allocates the geometry and its current matrix pointer would point at the object's absolute frame matrix in the hierarchy. The other 9 dynamic objects share the geometry buffers and have their matrix pointers pointing at the same frame matrix. Therefore, if you call this function, the matrix pointer of every dynamic object will point to the same matrix, the current position described by the frame. As discussed in the textbook, when actor references are used, we update the actor matrices first and then call the ObjectSetUpdated function so that the collision system can grab a snapshot of the current matrices. We then update the actor matrices for the second reference, call ObjectSetUpdated again, and so on. This is very important and has the potential to cause numerous problems if not remembered. There is no way the single actor can ever be in more than one pose at a time. If you were to call this function, all dynamic objects that were created from the same mesh in the same actor would all be assigned the same world matrix.

CCollision::Clear

There may be times when you wish to purge the collision system geometry database so that you can re-use the same collision object for a different task. The Clear method does just this. This method is also used by the CCollision destructor to release all memory before the system is deleted.

Clear resets the state of the CCollision object to its default state. It sets its internal transform matrix to an identity matrix and then loops through each dynamic object. For each dynamic object it finds that is not a reference, it will delete its geometry buffers. It does not do this for references as they do not own their own geometry. Whether a reference object or not, the dynamic object structure is then deleted from memory also. This is done for each dynamic object in the collision system's dynamic object vector.

```
void CCollision::Clear( )
{
    ULONG i;

    // Reset the internal transformation matrix
    D3DXMatrixIdentity( &m_mtxWorldTransform );
    // Release dynamic object pointers
    for ( i = 0; i < m_DynamicObjects.size(); ++i )
    {
```

```
        // Retrieve the object
        DynamicObject * pObject = m_DynamicObjects[i];

        // Delete if existing
        if ( pObject )
        {
            // We only delete our mesh data if were not a reference
            if ( !pObject->IsReference )
            {
                // Release the vectors
                if ( pObject->pCollVertices  ) delete pObject->pCollVertices;
                if ( pObject->pCollTriangles ) delete pObject->pCollTriangles;

            } // End if not reference

            // Delete the object
            delete pObject;
        }
    }
```

We then loop through each element in the collision system's terrain pointer vector. Before clearing this vector we must first call the Release method on each terrain pointer because our terrain object employs a COM style reference counting mechanism. You will recall how we incremented the reference count of a terrain object when its pointer was added in the AddTerrain method discussed earlier.

```
    // Release terrain objects
    for ( i = 0; i < m_TerrainObjects.size(); ++i )
    {
        // Retrieve the object
        CTerrain * pTerrain = m_TerrainObjects[i];

        // Release if existing
        if ( pTerrain ) pTerrain->Release();

    } // Next Terrain Object
```

Finally, we empty the static geometry vectors, the dynamic object vector, and the terrain vector, releasing all the memory they currently contain. We then reset the m_nLastObjectSet member variable back to its default state of -1 since there now no object sets registered with the collision system.

```
    // Empty our STL containers.
    m_CollTriangles.clear();
    m_CollVertices.clear();
    m_DynamicObjects.clear();
    m_TerrainObjects.clear();

    // Reset any variables
    m_nLastObjectSet = -1;
}
```

Registering Geometry with the Collision System

In this next section we will discuss the additions to the application's loading code that manage collision system registration. The CCollision object is actually a member of our CScene class. This is useful since this class manages loading geometry data, and the animation and rendering of that data in the main game loop. We will revisit functions such as CScene::ProcessMeshes, CScene::ProcessReferences, and CScene::ProcessEntities, which are no strangers to us. These are the functions we have used since the beginning of this training program to process the objects loaded from IWF files by the CFileIWF file object.

In this workbook we will focus on the additions to the IWF loading code in CScene. The CScene::LoadSceneFromX function will not be discussed since it has hardly changed from previous versions. It simply loads the single X file into a CActor object and then registers it with the collision system using the CCollision::AddActor function. Because adding actors to the collision system will be demonstrated in the IWF loading code, you should have no trouble noticing the necessary changes to this function.

Loading IWF Files – Recap

In virtually all of our previous projects we have used the CScene::LoadSceneFromIWF function to load in our geometry from IWF files. This function is called from the CGameApp::BuildObjects function which itself is called from CGameApp::InitInstance.

The LoadSceneFromIWF method uses the CFileIWF object contained in libIWF.lib. This library is part of the IWF SDK to automate the loading of IWF files. As we have discussed on previous occasions, the CFileIWF::Load function is used to load all the data objects contained in the IWF file into a number of vectors supplied by the library. For example, all entities contained in the IWF file are stored in the CFileIWF::m_vpEntityList vector and all internal meshes defined in the file are loaded into the CFileIWF::m_vpMeshList vector. Vectors also exist to store the materials and texture filenames. These vectors are automatically filled with data when CFileIWF::Load is called. This means all our scene object has to do on successful completion of this function is extract the data from these vectors and format it in the desired fashion. None of this is new to us as we developed this loading system way back in Module 1 and have been adding to it ever since.

The first updated function we will look at is the CScene::ProcessMeshes function. It is called from CScene::LoadSceneFromIWF to extract the data from the CFileIWF::m_vpMeshList vector. The function is passed a single parameter, the CFileIWF object that contains the meshes loaded from the file.

CScene::ProcessMeshes (Updated)

IWF files generally contain meshes in two different forms -- internal meshes and mesh references. If you have ever used GILES™ then you can think of the brushes that you place in the scene as the internal mesh type. The physical polygon data of such objects is saved out to the IWF file with a world matrix describing the position of the brush in the scene. In the case of GILES™, all brushes have their vertices

defined in world space, so the world matrix accompanying each mesh in the file will always be identity. The other type of mesh data that we have used is the reference entity. In GILES™, we can place a reference entity in the scene that contains the name of an X file. While GILES™ will physically render the geometry in the X file referenced by such an entity, this geometry is not saved out to the IWF file, only the filename of the X file is. The ProcessMeshes function is called to extract only internal geometry meshes whose geometry was stored in the IWF file. This geometry will have been loaded into the CFileIWF::m_vpMeshList array by the CFileIWF::Load function, so let us now parse this data.

We have discussed most of the code to this function before and as such we will simply step to the new code we have added.

In Lab Project 13.1, we decided only to add meshes to the collision database that are not flagged as detail objects (like a decoration), but you can easily change this behavior if you wish. After our mesh is constructed and the IWF copied, we call the collision system's AddIndexedPrimitive method and pass in the pointers to our new CTriMesh's vertex and index arrays. We use the member functions of CTriMesh to get all the information (vertex and face list pointers, the number of vertices and faces in those lists, and the stride of the vertex and indices). This single function call registers every single triangle of the CTriMesh with the collision system's static database. Notice that we do not bother setting the world transformation matrix because the internal meshes we load from GILES™ are already in world space.

```
        // Add this data to our collision database if it is not a detail object
        if ( !(pMesh->Style & MESH_DETAIL) )
        {
            // Add mesh data to collision database
            if ( !m_Collision.AddIndexedPrimitive(    pNewMesh->GetVertices(),
                                                      pNewMesh->GetFaces(),
                                                      pNewMesh->GetNumVertices(),
                                                      pNewMesh->GetNumFaces(),
                                                      pNewMesh->GetVertexStride(),
                                                      pNewMesh->GetIndexStride()))
            {
                // Clean up and fail, something bad happened
                delete pNewMesh;
                return false;

            } // End if failure to add data

        } // End if not detail object
```

CScene::ProcessReference

The CScene::LoadSceneFromIWF function calls the ProcessEntities function to process any entities that may have been loaded into the CFileIWF::m_vpEntities vector. External mesh references are stored as reference entities. The data area of a reference entity is arranged as a ReferenceEntity structure (CScene.h). This structure just contains the filename of the thing that it is referencing. When the ProcessEntities function determines it has found a reference entity, it calls the ProcessReference function. This function assumes that all reference entities are references to external X files and as such, the X files are loaded into actors along with any animation they may contain.

This function, while mostly unchanged, has a few new bits added. If the X file we are loading has already been loaded, then the actor is added to the collision system as an actor reference, otherwise it is added as a normal actor. We will move very quickly through this code, since most of it is familiar to us.

```
bool CScene::ProcessReference( const ReferenceEntity& Reference,
                               const D3DXMATRIX & mtxWorld )
{
    HRESULT                     hRet;
    CActor                    * pReferenceActor      = NULL;
    LPD3DXANIMATIONCONTROLLER   pReferenceController = NULL;
    LPD3DXANIMATIONCONTROLLER   pController          = NULL;
    long                        ObjectSetIndex       = -1;
    ULONG                       i;

    // Skip if this is anything other than an external reference.
    // Internal references are not supported in this demo.
    if (Reference.ReferenceType != 1) return true;
```

In the first section of code we test to see of the reference type member of the reference entity structure is set to 1 (external reference), if not we return as we do not currently support any other reference type. Notice how the ProcessEntities function will also pass in the world matrix of the reference which would also have been loaded from the file (as every entities is accompanied by a matrix in the IWF file).

Next we build the complete filename of the reference, and loop through each currently loaded actor. If we find an actor in the scene's CActor array that has the same name as the X file we are trying to load, then we know this X file has already been loaded into an actor and we break from the loop.

```
    // Build filename string
    TCHAR Buffer[MAX_PATH];
    _tcscpy( Buffer, m_strDataPath );
    _tcscat( Buffer, Reference.ReferenceName );

    // Search to see if this X file has already been loaded
    for ( i = 0; i < m_nActorCount; ++i )
    {
        if (!m_pActor[i]) continue;
        if ( _tcsicmp( Buffer, m_pActor[i]->GetActorName() ) == 0 ) break;

    } // Next Actor
```

We know at this point that if the loop variable *i* is not equal to the number of currently loaded actors, then the loop exited early because we found a matching actor. This means the X file geometry already exists in memory and we do not want to load it again. Instead, we will create an actor reference. We first get a pointer to the actor we are going to reference as shown below.

```
    // If we didn't reach then end, this Actor already exists
    if ( i != m_nActorCount )
    {
        // Store reference Actor.
        pReferenceActor = m_pActor[i];
```

We now see if any CObject exists in the scene's CObject array which is using the actor.

```
        // Find any previous object which owns this actor
        for ( i = 0; i < m_nObjectCount; ++i )
        {
            if (!m_pObject[i]) continue;
            if ( m_pObject[i]->m_pActor == pReferenceActor ) break;

        } // Next Object
```

If loop variable *i* is not equal to the number of objects in the CObject array, then we found an object that is using the actor we want to add to the collision system. Therefore, we will reference the object.

We first get a pointer to the CObject which contains the actor we wish to reference.

```
        // Add a REFERENCE dynamic object to the collision system.
        if ( i != m_nObjectCount )
        {
            CObject * pReferenceObject = m_pObject[i];
```

At this point we know that the CObject that already exists must have registered its actor with the collision system and as such, the CObject::m_nObjectSetIndex member will contain the object set index that was returned when this original actor was registered. We call the CCollision::AddActorReference function to register a new copy of this actor with the collision system at a different world space position. Notice how we pass in the object set index and the world matrix of the current reference we are processing. Provided the object set index we passed in exists, new dynamic objects will be created and added to the collision system as described by the matrix (the final parameter). These new dynamic objects (one for each mesh contained in the actor) will be added as a single new object set to the collision system and the index of that new set will be returned from the function.

```
            // Create a reference of this objects data in the collision system
            ObjectSetIndex = m_Collision.AddActorReference
                             ( pReferenceObject->m_nObjectSetIndex,
                               pReferenceObject->m_mtxWorld,
                               mtxWorld );
```

If the object we are referencing currently has no animation controller pointer, then it means no object references to this actor currently exist; it is only a single non-reference object. As soon as we add more than one CObject to the scene that uses the same actor, all CObjects that use that actor essentially become references and store their own animation controllers. We covered all of this logic before.

```
            // Is this the first reference?
            if ( !pReferenceObject->m_pAnimController )
            {
                // If this is the first time we've referenced this actor then
                // we need to detach its controller and store in the reference
                pReferenceObject->m_pAnimController =
                                        pReferenceActor->DetachController();

            } // End if first reference
```

```
            ULONG nMaxOutputs, nMaxTracks, nMaxSets, nMaxEvents;

            // Retrieve all the data we need for cloning.
            pController = pReferenceObject->m_pAnimController;
            nMaxOutputs = pController->GetMaxNumAnimationOutputs();
            nMaxTracks  = pController->GetMaxNumTracks();
            nMaxSets    = pController->GetMaxNumAnimationSets();
            nMaxEvents  = pController->GetMaxNumEvents();

            // Clone the animation controller into this new reference
            pController->CloneAnimationController( nMaxOutputs,
                                                   nMaxSets,
                                                   nMaxTracks,
                                                   nMaxEvents,
                                                   &pReferenceController );

        } // End if we found an original object reference.
```

The above section of code shows the conditional that happens when an object was found that uses the actor we wished to load. Outside that conditional code block, we test to make sure we have a valid object set index. If not, then it means that although the actor already exists (there is no need to create a new one, we can just reference it) it has ***not*** yet been assigned to a CObject or registered with the collision system. Therefore we cannot possibly register the actor with the collision system as a reference since it has not been added to the collision system in its non-referenced form. It is the first actor of this type we are registering.

When this is the case, we register the actor with the collision system as a normal actor. Notice that we register the actor with the collision system as a dynamic object group by passing false as the final parameter. Therefore, if this actor contains animation data, our application will be able to animate it and have the collision system respond to it in real time.

```
        // If the collision system could not match this object set index,
        // or we couldn't find an already existing object, add the full blown
        // actor.
        if ( ObjectSetIndex < 0  )
             ObjectSetIndex = m_Collision.AddActor( pReferenceActor,
                                                    mtxWorld,
                                                    false );

    } // End if Actor already exists
```

All the above code is executed if the file name of the external reference we are trying to load has already been loaded for a previous actor. In short, if it has, we decide we never want two copies of the same frame hierarchy in memory, so we reference it both in the scene and in the collision system.

The next section of code shows what happens when the actor has not already been loaded. When this is the case, we must create a new actor, load the X file and register its attribute callback function (CScene::CollectAttributeID). We then load the actor from the X file. The name of the X file we wish to load is the name of the external reference which is now stored in the Buffer array along with the data path.

```
    else

    {
        // Allocate a new Actor for this reference
        CActor * pNewActor = new CActor;
        if (!pNewActor) return false;

        // Load in the externally referenced X File
        pNewActor->RegisterCallback( CActor::CALLBACK_ATTRIBUTEID,
                                     CollectAttributeID, this );

       HRet = pNewActor->LoadActorFromX(  Buffer,
                                          D3DXMESH_MANAGED,
                                          m_pD3DDevice );

       if ( FAILED(hRet) ) { delete pNewActor; return false; }
```

With the actor now loaded, we make room at the end of the scene's CActor array for another actor pointer and store our current actor. Our code upgrade (in bold) now registers this new actor with the collision system. We also assign the local pRefereneActor pointer to point at this actor so that outside this conditional code block we can use the same pReferenceActor pointer whether a new actor was loaded, or whether it just points to an actor that was previously loaded.

```
        // Store this new Actor
        if ( AddActor( ) < 0 ) { delete pNewActor; return false; }

        m_pActor[ m_nActorCount - 1 ] = pNewActor;

        // Add the physical actor to the collision system
        ObjectSetIndex = m_Collision.AddActor( pNewActor, mtxWorld, false );

        // Store as object reference Actor
        pReferenceActor = pNewActor;

    } // End if Actor doesnt exist.
```

At this point, regardless of whether we created and loaded a new actor or are using one that was already loaded, pReferenceActor will point at it. Let us now create a new CObject to hold this actor. Notice again that we pass the actor pointer into the CObject constructor.

```
    // Now build an object for this Actor (standard identity)
    CObject * pNewObject = new CObject( pReferenceActor );
    if ( !pNewObject ) return false;
```

We also store the world matrix of the reference (passed into the function) and a pointer to the animation controller it will use. Along the way we will store a copy of the object set index that was assigned during actor registration with the collision system. Finally, we add this object to the end of the scene CObject array and return.

```
    // Copy over the specified matrix and store the colldet object set index
    pNewObject->m_mtxWorld        = mtxWorld;
```

```
    pNewObject->m_nObjectSetIndex = ObjectSetIndex;

    // Store the reference animation controller (if any)
    pNewObject->m_pAnimController = pReferenceController;

    // Store this object
    if ( AddObject() < 0 ) { delete pNewObject; return false; }
    m_pObject[ m_nObjectCount - 1 ] = pNewObject;

    // Success!!
    return true;
}
```

CScene::ProcessEntities

The process entities function has had one line added to it to register any loaded terrains with the collision database. We will not show the entire function but only a subsection of the code that deals with the terrain entity type. The complete loading code for the terrain entity was discussed in the previous lesson. We start at the switch statement used to determine which type of entity we are processing.

```
            switch ( pFileEntity->EntityTypeID )
            {
                  …
                  …
                  Code Snipped here for other entity types
                  …
                  …

                   case CUSTOM_ENTITY_TERRAIN:

                        …
                        … Code snipped here which copies entity info
                        … in to terrain entity structure Terrain

                        // Allocate a new terrain object
                        pNewTerrain = new CTerrain;
                        if ( !pNewTerrain ) break;

                        // Setup the terrain
                        pNewTerrain->SetD3DDevice( m_pD3DDevice, m_bHardwareTnL );
                        pNewTerrain->SetTextureFormat( m_TextureFormats );
                        pNewTerrain->SetRenderMode(GetGameApp()->GetSinglePass());
                        pNewTerrain->SetWorldMatrix
                                    ((D3DXMATRIX&)pFileEntity->ObjectMatrix );
                        pNewTerrain->SetDataPath( m_strDataPath );

                        // Store it
                        m_pTerrain[ m_nTerrainCount - 1 ] = pNewTerrain;

                        // Load the terrain
                        if ( !pNewTerrain->LoadTerrain( &Terrain ) ) return false;

                        // Add to the collision system
```

```
                        m_Collision.AddTerrain( pNewTerrain );

                    } // End if standard terrain

                    break;
```

In the source code shown above, we have also removed all the code that simply copies the terrain data of the terrain from the CFileIWF object into a terrain entity structure. It still exists in the source code but has been removed here for readability.

After doing all of our usual setup for managing the terrain, as a final step, we call the CCollision::AddTerrain method to add a pointer to this terrain object to the end of the collision object's terrain list. As we have seen, the collision system can handle collisions with such terrain objects in a memory efficient way because it does not need to store a complete copy of every terrain triangle.

Updating the Geometry Database at Runtime

You should be well aware by now that the heartbeat of our application is the CGameApp::FrameAdvance function. At a high level, our runtime processing comes down to repeatedly calling this function in a loop. This function controls what happens for every single frame of the game and controls the flow in which events are processed. For example, each time it is called, it instructs the application to process any input that may have been given by the user. It then instructs the scene to apply any animations to any of its objects that it wishes for the current frame update. It also instructs the camera to update its view matrix and finally, it instructs both the scene and the CPlayer objects to render themselves.

Below we see the main snippet of the CGameApp::FrameAdvance function. We have not shown the code that tests for and recovers lost devices or the code that presents the back buffer. Instead we see all of the function calls made to transform and render the scene and the order in which states are updated in a given iteration of the game loop.

```
    // Poll & Process input devices
    ProcessInput();

    // Animate the scene objects
    m_Scene.AnimateObjects( m_Timer );

    // Update the Player ( Used to be in ProcessInput )
    m_Player.Update( m_Timer.GetTimeElapsed() );

    // Update the device matrix
    m_pCamera->UpdateRenderView( m_pD3DDevice );

    // Clear the frame & depth buffer ready for drawing
    m_pD3DDevice->Clear(0,NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0x79D3FF,
                        1.0f, 0 );

    // Begin Scene Rendering
    m_pD3DDevice->BeginScene();
```

```
    // Render the scene
    m_Scene.Render( *m_pCamera );

    // End Scene Rendering
    m_pD3DDevice->EndScene();
```

Notice in the above code snippet from Lab Project 13.1 that we now have a CPlayer::Update call after the call to AnimateObjects. This function call used to update the position of the player was originally called directly from the ProcessInput function. We will discuss why we have moved it out of that function and into the FrameAdvance function shortly.

The above code clearly shows that the CScene::AnimateObjects function is called during every iteration of the game loop. This gives the scene a chance to update the matrices of any scene objects it wishes to animate. It also allows the scene object to advance animation controllers of any actors currently participating in the scene. We need to revisit this function so that we can add the code that will notify the collision system when the scene geometry has been animated.

CScene::AnimateObjects

In this lesson we will have to inform the collision system when any objects that have been registered as dynamic objects with the collision system have been animated.

The function first loops through every CObject in the scene's CObject array. In Lab Project 13.1, we only animate actors, so we are not interested in finding any objects in this function which contain only a single CTriMesh. You can see that at the start of the object loop, we skip the current object if it does not have a valid CActor pointer.

```
void CScene::AnimateObjects( CTimer & Timer )
{
    ULONG i;

    // Process each object for coll det
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CObject * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Get the actor pointer
        CActor * pActor = pObject->m_pActor;
        if ( !pActor ) continue;
```

We next test to see if the object we are about to animate has a valid animation controller pointer. If the pointer is NULL, then it means either the actor has no animation to play, or this is the only object that references the actor and therefore the animation controller is owned by the actor itself. If the object does have a valid animation controller pointer, then there are multiple objects that reference this object's actor. This means we must attach the animation controller of this reference object to the actor so that we can animate the hierarchy using the reference's animation data. Once we attach the controller to the

actor, we advance its timeline. This will cause the parent relative frame matrices of the actor's hierarchy to be rebuilt in the pose described by the reference's animation controller. The parent relative matrices of the hierarchy will now describe the actor exactly as it should look for this particular reference.

```
        if ( pObject->m_pAnimController )
            pActor->AttachController( pObject->m_pAnimController, false );

        // Advance time
        pActor->AdvanceTime( Timer.GetTimeElapsed(), false );
```

Next we get the object set index assigned by the collision system when it was registered during the CScene::ProcessReference function. We set the actor's world matrix and pass true to the CActor::SetWorldMatrix function which forces all the absolute matrices of the actor to be rebuilt in their correct world space positions. Remember, it is these absolute matrices that the dynamic objects in the collision system point to. We must remember to update the world space matrices *before* instructing the collision system to update the status of the object group.

```
        //If the object has a collision object set index,
        // update the collision system.
        if ( pObject->m_nObjectSetIndex > -1 )
        {
            // Set world matrix and update combined frame matrices.
            pActor->SetWorldMatrix( &pObject->m_mtxWorld, true );

            // Notify the collision system that this set of dynamic objects
            // positions, orientations or scale have been updated.
            m_Collision.ObjectSetUpdated( pObject->m_nObjectSetIndex );

        } // End if actor exists

    } // Next Object

}
```

Once the world matrices of the actor have been built, we call the CCollision::ObjectSetUpdated function, passing in the object set index for the reference. This will instruct the collision system to search for all dynamic objects that were spawned from this actor and recalculate their collision and velocity matrices based on the updated world space frame matrices.

Collision Geometry Management – Final Note

We have now covered everything we need to know about registering scene geometry of different types with our collision system. We have seen how to load and register static meshes, actor references, and terrain entities with the collision system. We have also talked about how we should correctly manage updates for scene geometry that is registered as dynamic.

What we have not yet discussed are the actual moving entities that will use our collision system to collide and slide in the environment. In Lab Project 13.1, we will use the CPlayer object as the moving

entity. It will support both a first and third person camera. The mesh of the player in first person mode will be our U.S. Army Ranger from the prior lessons on skinning.

By attaching the skinned mesh to the CPlayer object, we can move it about the scene and watch it collide and slide in the environment. However, our CPlayer object has always been a little rudimentary when it comes to its physics handling. Now we will need to pay a little more attention to these details because we want to implement player movement that is typical in a first/third person game. Therefore, in the next section, we will discuss the changes to our player class to see how some simple improvements to our physics model will provide better environment interaction in our collision system.

> **Note**: Our intent is not to design a proper physics engine in this course. That would be the subject of an entire course by itself. We are simply trying to provide a more realistic feel for our player as it navigates throughout the world. Although the system we use is based on some laws of Classical Mechanics, it is still going to be a very rudimentary system. Learning how to create an actual physics engine with support for rigid bodies, vehicles, and the like, falls into the domain of the Game Physics course offered here at Game Institute.

The Revised CPlayer Class

Our previous CPlayer class was pretty straightforward and much of the relationship between the CGameApp object and the CPlayer object will remain intact. However, the CPlayer::Update function, which was responsible for calculating the player velocity will be totally rewritten. For one thing, it will use more appropriate physics calculations. It will also issue an update request to the collision system via a CScene callback function.

Let us quickly recap the way things used to work so that we get a better idea for what has to change.

- The CGameApp::FrameAdvance function would call the CGameApp::ProcessInput function with each iteration of the game loop.

- The ProcessInput function would read the state of the keyboard to combine a number of CPlayer flags describing the user direction request. The mouse would also be read to determine if the user wished to rotate the view.

- If the player tried to move, the ProcessInput function would call the CPlayer::Move function passing in the direction(s) the player should move (using a combination of flags) and the distance to move. The CPlayer::Move function would build a vector pointing in the requested direction with the requested length and add it to the current velocity vector (maintained internally). This updated velocity vector would not yet be used to update the position of both the player object itself and its attached camera. That happened later. All we have done this point is added a force to the velocity vector.

- If the ProcessInput function determined that the user also wished to rotate the player object (via mouse input), it would call the CPlayer::Rotate function to perform a local rotation to the CPlayer's world matrix.

- At this point in the ProcessInput function, the player object has been potentially rotated and its velocity vector has been updated, but the position has not yet been altered based on that velocity. The ProcessInput function would finally call the CPlayer::Update function before returning to apply the movement described by the velocity vector that had just been updated.

- The CPlayer::Update function is the one that will require the most changes in this new version. Previously, it added a gravity vector to the player's velocity vector so that a constant downwards force was applied to the player each frame. This would make sure that if the player had no terrain underneath him (the only type of collidable geometry we supported), they would fall downwards. We would also downscale the velocity vector of the player each frame based on a friction constant. By shortening the length of the velocity vector based on friction each time this function is called, we allowed our player to slow to a halt when the user released the movement keys instead of carrying on forever, as would be the case in a frictionless environment. The larger the value we set for the player object's friction, the more suddenly it would stop when no movement was being applied by the user. This is because the velocity vector will be shortened the next time the update function is called as long as the user does not press another movement key.

- The CPlayer::Update function would update the actual position of the player using the velocity vector. This would physically move the player object in the world.

- Keep in mind that the player object may have updated its position with respect to another scene object. You will recall that the CPlayer object maintained an array of callback functions to help in this regard. In previous applications, the CTerrain class registered a callback function with the player. This callback was invoked from the CPlayer::Update function after the position of the player had been updated. The callback function had a chance to examine the new position of the player and modify it if it finds it improper. In our previous applications, the terrain callback function would test the position of the player against the terrain geometry. If it found that the position of the player had been moved below the height of the terrain at that location, it would modify the height of the player's position so that it sat on top of the terrain. This is what prevented our simple gravity model from pushing the player through the terrain. This callback function will now be replaced with a CScene callback function which will use our new collision system.

- As the CPlayer::Update function also updated the position of the camera, it also instructed the camera to call any callback functions which have been registered for it. This allowed the same collision detection function to be used to modify the position of the player and its attached camera before returning.

- The CGameApp::FrameAdvance function would also call the CPlayer::Render function each frame to instruct the player to render any attached mesh (such as the third person mesh).

Our CPlayer object will now have the following member variables. Not all of them are new to us since our player class has always maintained a velocity vector, a gravity vector, and a scalar used to store the amount of drag to apply to the camera position when tracking the player in third person mode. Some of the other members shown below will be new and their usefulness will be described in this section.

Except from CPlayer.h (Lab 13.1)

```
    // Force / Player Update Variables
    D3DXVECTOR3     m_vecVelocity;          // Movement velocity vector
    D3DXVECTOR3     m_vecAppliedForce;      // Our motor force
    D3DXVECTOR3     m_vecGravity;           // Gravity vector
    float           m_fCameraLag;           // Amount of camera lag in seconds
    float           m_fTraction;            // How much traction we can apply
    float           m_fAirResistance;       // Air resistance coefficient.
    float           m_fSurfaceFriction;     // Fake Surface friction scalar
    float           m_fMass;                // Mass of player
```

There are also some simple inline public member functions to set the above properties of the player.

```
void SetGravity  ( const D3DXVECTOR3& Gravity  ) { m_vecGravity = Gravity; }
void SetVelocity ( const D3DXVECTOR3& Velocity ) { m_vecVelocity = Velocity; }
void SetCamLag   ( float CamLag )                { m_fCameraLag = CamLag; }
void SetTraction ( float Traction              ) { m_fTraction = Traction; }
void SetSurfaceFriction ( float Friction       ) { m_fSurfaceFriction = Friction; }
void SetAirResistance   ( float Resistance     ) { m_fAirResistance = Resistance; }
void SetMass            ( float Mass           ) { m_fMass = Mass; }
```

Useful Concepts in Classical Dynamics

Before we start examining the implementation of our application's physics model, let us begin with a very high level overview of some of the physics concepts we would like to consider as we put together our new system. To be sure, this will be a very quick introduction to Classical Dynamics. For a much more complete discussion, it is highly recommended that you take the Game Physics course offered here at the Game Institute.

Newton's Laws of Motion

We will start with some very basic ideas: Newton's Three Laws of Motion.

First Law: *When the net force on an object is zero, the motion of an object will be unchanged. An object at rest will remain so, unless compelled to change because some amount of force is applied.*

The first law basically tells us that if an object is moving with some direction and magnitude (i.e., velocity), it will continue moving with that velocity unless some outside force acts on it causing some change.

Second Law: $\sum \boldsymbol{F} = m\boldsymbol{a}$

The second law talks about what happens to an object when forces are applied. As we can see, the law gives us a relationship between the net forces acting on the object, the object's acceleration, and the mass of that object. In other words, using this law, we can eventually figure out how fast an object will go based on how much mass it has and how much force is applied to it.

Note that in the above formula, force and acceleration are both vector quantities. Thus, they act along all three axes in the case of a three dimensional system.

It is also worth noting that the unit of measurement for force is called a newton (1 kg * m / s^2), often given by the letter N. The units of measurement might seem a little strange at first due to the s^2 concept in the denominator. But this makes more sense when you recall the relationship between position, velocity, and acceleration. We know that velocity represents a change in position with respect to time. This basically tells us how fast we are going (our speed).

$$\mathbf{v} = \Delta\mathbf{p} / \Delta t$$

Acceleration tells us how much our velocity is changing with respect to time.

$$\mathbf{a} = \Delta\mathbf{v} / \Delta t$$

This is where we see our unit for acceleration come into play. If we substitute in our equation for velocity, we get:

$$\mathbf{a} = (\Delta\mathbf{p} / \Delta t) / \Delta t$$

$$\mathbf{a} = (\Delta\mathbf{p} / \Delta t^2)$$

Since position is measured in meters and time is measures in seconds, we get our acceleration units of meters per second per second (or, given the rules of fractional division: m / s^2). When we factor in mass, which is measured in kilograms, we wind up with our newton as described above. Thus, one newton is the amount of force required to give an object with a mass of 1 kilogram an acceleration of 1 meter per second per second.

Third Law: *For every action, there exists an equal and opposite reaction.*

The third law gives us a relation between the forces that exist between two interacting bodies. What it basically states is that when an object A applies a force to another object B, object B is applying the same force to object A, just in the opposite direction. Mathematically, we can state this relationship as:

$$\mathbf{F}_B = -\mathbf{F}_A \quad \text{or} \quad \mathbf{F}_A + \mathbf{F}_B = 0$$

Thus when I am in contact with the ground, while gravity might force me downwards towards the center of the earth, the ground exerts an opposite upwards force that is equal and opposite and I remain in place on the surface. I do not get shoved through the surface, nor do I hover above the ground. I am in equilibrium when I am in contact with the surface.

Contact Forces

Now let us talk a little bit about the forces that come into play when objects are moving about in the environment.

On Earth, all solid bodies experience resistance to motion. Whether they are sliding on solid surfaces, rolling along on those surfaces, or moving through a liquid or gas, some amount of resistance will be given. For our purposes in this lesson, we will consider the general concept of resistance as falling within the domain of *friction*.

Friction forces depend on the types of surface that are in contact. Generally, the rougher the surface, the greater the friction. We can generally break down the concept of friction into two categories: *static friction* and *dynamic* (or *sliding*) *friction*. Static friction is essentially the amount of force that must be overcome before an object begins to slide. Once an object is in sliding, dynamic friction comes into play. Dynamic friction tells us how much force needs to be overcome in order to keep our object sliding along on that surface.

Consider the example of an automobile. There exists a certain amount of static friction between the rubber of the car's tire and the asphalt of the road at the point where the two come into contact. As long as the forces that are being applied to the tire (and thus the contact point) do not exceed this static friction threshold, the car tire is able to grip the road and propel the car forward. As the tire spins, it pushes down on the road, and because of static friction, the road pushes back on the tire (see Newton's Third Law) and the car continues it forward motion. But if the car were to suddenly hit a patch of ice, where the static friction between rubber and ice is significantly lower, the static friction hurdle would be much easier to overcome. If the forces were such that they exceeded the static friction threshold, the tire

would begin to slide. Although the tire is still spinning, it is not be able to get a good purchase on the surface and suddenly dynamic friction (sliding friction) comes into play.

Friction is ultimately proportional to the total amount of force pressing the objects together. In most cases, this will be the force due to gravity (a downwards acceleration) which we can describe as:

$$\mathbf{F}_G = m\mathbf{g}$$

Where m is the mass of the object and **g** is the gravitational acceleration (9.8 m/s^2). Note the use of Newton's Second Law as a means to describe gravity.

According to Newton's Third Law there must be a net equal and opposite force that counters this one. That is, if gravity forces a body downwards, assuming the body is in contact with a surface, there must be a force pressing back upwards on the bottom of the object. There is indeed such a force, and it is generally referred to as the *normal force* (**N**). The normal force is directed based on the orientation of the surface.

$$\mathbf{N} = -m\mathbf{g}$$

The proportionality between friction and the normal force can be expressed by a constant which can be introduced as a coefficient. This coefficient is represented by the Greek letter mu (μ). Thus:

$$\mathbf{F}_{(\text{sliding friction})} = \mu\mathbf{N}$$

We can substitute in our gravity values for the normal force and rewrite the equation as:

$$\mathbf{F} = -\mu\, m\mathbf{g}$$

In the case of static friction, we can describe this force using the formula:

$$\mathbf{F}_s \leq \mu_s\, m\mathbf{g}$$

μ_s is the coefficient of static friction that is associated with the two types of materials in contact. In other words, the static friction force remains below a certain threshold given by the normal force scaled by some constant.

Dynamic friction can be described by the very similar formula

$$\mathbf{F}_D = \mu_d\, m\mathbf{g}$$

Once again, we see the relationship with the normal force scaled by some constant. The following table provides some friction coefficient values that have been determined by experiment for various materials.

Material 1	Material 2	Static μ_s	Dynamic μ_d
Steel	Steel	0.74	0.57
Aluminum	Steel	0.61	0.47
Copper	Steel	0.53	0.36
Brass	Steel	0.51	0.44
Zinc	Cast Iron	0.85	0.21
Copper	Cast Iron	1.05	0.29
Glass	Glass	0.94	0.4
Rubber	Concrete (dry)	1	0.8
Rubber	Concrete (wet)	0.3	0.25

Not surprisingly, the coefficients of static friction turn out to be higher than those for dynamic friction. As you own experience tells you, it is generally easier to slide a heavy box along your kitchen floor after you have "broken it loose" from the grip of static friction.

> **Note:** The forces that exist when two bodies are touching are called *contact forces*. Thus friction and the normal force are both contact forces. The normal force is a perpendicular force that the surface exerts on the body it is in contact with. The friction force is a parallel force that exists in the direction of motion on the surface.

We used the example of an automobile earlier to introduce the concepts of static and dynamic friction. In practice, for objects that are rolling, a third type of frictional coefficient can be introduced called the *coefficient of rolling friction* (μ_r). This is sometime referred to as *tractive resistance*. The formula is pretty much the same as we see in all of our friction cases:

$$\mathbf{F} = \mu_r \mathbf{N}$$

For steel wheels on steel rails, μ_r generally equals 0.002. For rubber tires on concrete, μ_r equals ~0.2. Thus we see that the steel wheel/rail case experiences less resistance to movement than does the rubber/concrete case. This actually gives some indication as to why trains are typically more fuel efficient than automobiles.

> **Note:** We often hear the term *traction* used to describe this friction relationship as the amount of 'grip' that a wheel or tire has on the surface. The car tire on the asphalt surface has very good traction (a good grip), while the same tire on a sheet of ice has very poor traction (possibly no grip at all).

Fluid Resistance

We now understand that all bodies on Earth experience resistance to motion and we have seen friction at work with respect to movement on solid surfaces. But what about fluids (liquids and gases)? Certainly we all recognize from experience that moving through a pool of water is generally much more difficult than walking down the street. So there is obviously some manner of resistance happening there. Is this friction as well? Yes indeed.

Viscosity is a term used to describe the thickness of a fluid. Thicker fluids have higher viscosity and vice versa. Objects moving through high viscosity fluids will experience a greater amount of friction than similar movement through low viscosity fluids. However, as it turns out, when an object moves through a viscous fluid (including air), the resistance force also takes into account the speed of the object. To be fair, this will actually vary from one situation to the next and we are going to oversimplify things here a little bit, but in general, for slow moving objects we find that:

$$\mathbf{F}_R = -k\mathbf{v}$$

Where **v** is the velocity of the object and *k* is a constant (called the *coefficient of frictional drag*) that varies according to the size and shape of the body, as well as other factors having to do with the viscosity of the fluid.

For objects moving at higher rate of speed the formula becomes:

$$\mathbf{F}_R = -k\mathbf{v}^2$$

So as the speed of the object increases, so does the amount of resistance forces exerted on it by the fluid. This is the result of turbulence in the fluid due to the rapid movement of the object. The result is a higher level of friction between the moving object and the fluid.

If you were modeling slow moving vehicles in your game (cars, ships, etc.) then you would preferably use the first formula to model fluid resistance (often called *aerodynamic drag*, or just *drag*). If you were modeling high speed moving objects (racing cars, jet-skis, etc.) then the second formula would produce more accurate results.

Updating the CPlayer Class

At this point we have now covered all of the topics we will need to know about when we are updating our application's physics model. To be clear right upfront, we are *not* going to model precision physics that obey all of the formulas we have just discussed. Often it is not just practical or necessary to do so when working on a game. However, we will absolutely use the ideas that were presented as a basis for trying to simulate behavior that we find satisfying in our particular demo. We will attempt to maintain the spirit of a particular concept, but not necessarily follow the specific rules and equation. As we introduce concepts in our model, we will talk about how they relate to the theory just presented and, when we make some alterations of our own, explain why we did so.

While our new CPlayer object will work in a very similar manner to its prior incarnation, we will need to alter the CPlayer::Update function. But before we look at the code to the new CPlayer object, let us talk a little bit about the model on a high level and look at some of the member variables that were introduced to support that model. This should help set the stage for what is to come on the coding side.

Motor Force and Tractive Force (m_vecAppliedForce / m_fTraction)

Motor force is the amount of force that is applied to an object to make it move. If we think about the soles of a person's shoes in contact with the ground, when we apply motor force to the leg, that leg pushes on the ground with an applied force and in response, the ground pushes back in the opposite direction with its own force, essentially propelling the shoe (and thus the person) forward. We talked about some of these forces in the last section. For example, we know about the force pushing upwards from the ground (we called it the normal force) and we also know that some degree of resistance comes into play (friction). Our model will essentially take some liberties and combine these concepts together and we will call the resulting force the *tractive force*. That is, this tractive force will be the combination of the applied motor force(s) and the normal force (technically, the inverted gravitational force), with a dash of friction thrown in for good measure. We will discuss the friction model in more detail later. For now, the best way to understand our tractive force as it relates to friction is to think about the concept of traction introduced earlier. That is, we are going to say that there is a 'grip' factor that comes into play with respect to our surfaces as we attempt to apply accelerating forces. This will prove to be a very helpful concept when we model jumping a bit later on.

The amount of tractive force produced will depend on a traction coefficient for the surface (m_fTraction in our model). We can think of the traction coefficient as describing how good a grip the player's shoe will have with the surface and we will use a value in the range [0.0, 1.0]. The closer to 1.0 the traction coefficient is, the better the grip of the shoe on the surface. Since we will use this coefficient to scale the motor force, larger coefficient values mean the closer to the initially applied motor force the tractive force applied to the object will ultimately be. We can think of the motor force as being the force describing how much we wish to move, while the tractive force describes the amount we will actually move, once slippage between the surface and the object is taken into account. Again, gravity will play a role as well, but we will talk more about that shortly.

So our traction coefficient is actually going to act very much like a frictional coefficient in that it will scale the resulting force based on the properties of the contact surfaces. Consider an Olympic athlete's running shoes, which are specifically designed to get as near to perfect traction with the running track as possible. If we imagine the same athlete running on ice, the same amount of motor force would still be getting applied by the athlete, but because of the lack of grip between his/her shoes and the ice, the amount of tractive force actually pushing the athlete forward is going to be much less, and as such, slower running speeds are observed.

So a traction coefficient for an interacting surface/object pair of 0.5 would describe quite a slippery situation, since only half the applied motor force will be applied to the object in the forward direction.

Traction = 0.5

Let us also assume that we wish to apply a motor force of 20 to the object to move it along its velocity vector.

Applied Force = 20N

The amount of force that actually gets applied (the *tractive force*) is calculated as follows:

Tractive Force = Applied Force * Traction
= 20N * 0.5
= 10N

Again, for the moment, we are not including gravity in our model. We will address that a bit later.

In this example, you can see that although we applied a force of 20N to the object, the actual forwards force applied to the object was only half of that because of the lack of grip. As mentioned earlier, this is why traction is so important between the wheels of a racing car and the track on which it is racing. Racing cars often use tires that do not have treads in order to allow more rubber to come into contact with the surface of the track and provide a better grip for high speed racing.

Although the relationship between the player and the various surfaces in the level in real life would introduce a lot of different traction coefficients between surface types, we will simplify and assign our CPlayer object a single traction coefficient. This means, the traction between the player object and every surface in our level will be considered to be exactly the same. Obviously this could easily be extended so that a traction coefficient could be assigned at the per polygon level. That way, you could assign a polygon with an ice texture map a low traction of 0.2 but assign a much higher traction coefficient to a polygon with an asphalt texture mapped to it. In this demo, we will use a single traction coefficient for a given player object which will be set and stored in the CPlayer::m_fTraction variable shown above. The traction coefficient used by the player for its physics calculations can be set by the application using the CPlayer::SetTraction method.

Note that we will also be modeling dynamic friction, so you might wonder why we would need this traction concept if we intended to use proper friction anyway. As it turns out, in actual practice, our traction coefficient will almost always equal 1.0. The only time this changes (and drops to nearly zero) is when the player is no longer in contact with a surface (i.e., they are in the air). The idea was to give the player a little bit of ability to control the player while they are in mid-air. Since a person cannot actually walk on air, the traction concept gives them just a little bit control that they would not otherwise have using a standard model. This type of mid-air control is fairly common in games, so we decided to include it in our demonstration as well.

Applying motor force to our CPlayer object will be the primary means by which the application controls the speed of its movement. The CPlayer object will now have an ApplyForce method which allows the application to apply a motor force to the player. You will see later when we revisit the code to the CGameApp::ProcessInput function, that in response to keys being pressed by the user, we no longer call the CPlayer::Move function. Instead, we will call the CPlayer::ApplyForce method which adds motor force to the object. The more motor force we apply, the faster our object will move (assuming that our traction coefficient is not set to zero).

The CPlayer::ApplyForce function is shown below:

```
void CPlayer::ApplyForce( ULONG Direction, float Force )
{
    D3DXVECTOR3 vecShift = D3DXVECTOR3( 0, 0, 0 );

    // Which direction are we moving ?
```

```
    if ( Direction & DIR_FORWARD  ) vecShift += m_vecLook;
    if ( Direction & DIR_BACKWARD ) vecShift -= m_vecLook;
    if ( Direction & DIR_RIGHT    ) vecShift += m_vecRight;
    if ( Direction & DIR_LEFT     ) vecShift -= m_vecRight;
    if ( Direction & DIR_UP       ) vecShift += m_vecUp;
    if ( Direction & DIR_DOWN     ) vecShift -= m_vecUp;

    m_vecAppliedForce += vecShift * Force;
}
```

As you can see, ApplyForce accepts two parameters. The first is a 32-bit integer containing a combination of one or more flags describing the direction we wish to move the object. Remember, this function is called from the CGameApp::ProcessInput function, so the direction flags that are set correspond precisely to the direction keys being pressed by the user.

> **Note:** The movement flags such as DIR_FORWARD and DIR_UP are members of the Direction enumerated type that has been part of the CPlayer namespace since Chapter 4.

The function calculates a shift vector which describes the direction we wish to move the object. For example, if the forward key is pressed, then the player object's look vector is added to the shift vector. If both the up and forwards keys are pressed, then the shift vector will be a combination of the player object's up and look vectors, and so on. The result is a unit length shift vector describing the direction we wish to move the player. We then scale this vector by the passed motor force parameter before adding it to the m_vecAppliedForce member variable. Since forces are vector quantities that have both a direction and magnitude, you can see that in this particular case, we wind up with the proper result (just split over two logical code sections for ease of implementation, where the magnitude is passed in separately).

When we multiply the unit length direction vector by the motor force scalar in the above code, we generate a vector whose direction represents the direction we wish to move and whose length represents the amount of force we wish to apply to the object to move it in that direction (the motor force). Notice that we do not assign this value to the m_vecAppliedForce member variable but instead add it. Why?

The m_vecAppliedForce vector describes the direction and amount we wish to move. It contains the motor force that has been applied since the last frame. This is very important because you will see in a moment that this member is always set to a zero vector again after each CPlayer::Update call. Therefore, if it is always zero at the start of each iteration of our game loop, why are we not assigning (vecShift * Force) instead of adding it? The answer is that other places in the application may wish to apply forces to the player object in a given iteration of the game loop. As such, we will collect all forces that have been applied in this update. When we do eventually call the CPlayer::Update method to apply m_vecAppliedForce and generate the player's current velocity vector and calculate a new position, it will contain the combination of all forces applied since the last update.

We still have a ways to go yet, but now that we have seen the new ApplyForce method, let us quickly revisit the CGameApp::ProcessInput function. This is called at the start of the update process for each frame. It is called by the CGameApp::FrameAdvance function.

CGameApp::ProcessInput

Most of this function is unaltered so we will quickly skip past the bits we have already discussed in previous lessons.

```
void CGameApp::ProcessInput( )
{
    static UCHAR pKeyBuffer[ 256 ];
    ULONG        Direction = 0;
    POINT        CursorPos;
    float        X = 0.0f, Y = 0.0f;

    // Retrieve keyboard state
    if ( !GetKeyboardState( pKeyBuffer ) ) return;

    // Check the relevant keys
    if ( pKeyBuffer[ VK_UP    ] & 0xF0 ) Direction |= CPlayer::DIR_FORWARD;
    if ( pKeyBuffer[ VK_DOWN  ] & 0xF0 ) Direction |= CPlayer::DIR_BACKWARD;
    if ( pKeyBuffer[ VK_LEFT  ] & 0xF0 ) Direction |= CPlayer::DIR_LEFT;
    if ( pKeyBuffer[ VK_RIGHT ] & 0xF0 ) Direction |= CPlayer::DIR_RIGHT;
    if ( pKeyBuffer[ VK_PRIOR ] & 0xF0 ) Direction |= CPlayer::DIR_UP;
    if ( pKeyBuffer[ VK_NEXT  ] & 0xF0 ) Direction |= CPlayer::DIR_DOWN;
```

The first section of code (shown above) uses the GetKeyboardState function to read the state of each key into the local 256 byte buffer. The state of each key we are interested in (Up, Down, Left, Right, PageUp and PageDown) is checked to see if it is depressed. If so, the corresponding CPlayer::Direction flags are combined into the 32-bit variable Direction. At this point the Direction variable has a bit set for each direction key that we pressed.

Next we see if the mouse button is currently being held down, in which case our application window needs to be managing the capture of mouse input. If so, we get the current position of the mouse cursor and subtract from it the previous position of the mouse cursor for both the X and Y axes. Dividing these results by 3 (arrived at by trial and error) gives us our pitch and yaw rotation angles. We then reset the mouse cursor back to its previous position so that it always stays at the center of the screen. We must do this because if we allowed the cursor to physically move to the edges of the screen, the player would no longer be able to rotate in that direction when the operating system clamps the mouse movement to the screen edges.

```
    // Now process the mouse (if the button is pressed)
    if ( GetCapture() == m_hWnd )
    {
        // Hide the mouse pointer
        SetCursor( NULL );

        // Retrieve the cursor position
        GetCursorPos( &CursorPos );

        // Calculate mouse rotational values
        X = (float)(CursorPos.x - m_OldCursorPos.x) / 3.0f;
        Y = (float)(CursorPos.y - m_OldCursorPos.y) / 3.0f;
```

```
        // Reset our cursor position so we can keep going forever :)
        SetCursorPos( m_OldCursorPos.x, m_OldCursorPos.y );

    } // End if Captured
```

The next section of code is only executed if either the player needs to be moved or rotated. If the X and Y variables are not equal to zero then they describe the amount of pitch and yaw rotation that should be applied to the player. First we will deal with the rotation.

If the right mouse button is being held down then the left and right mouse movement should apply rotations around the player object's local Z axis (roll). Otherwise, the left and right movement should apply rotations around the player object's Y axis (yaw). In both cases, the backwards and forwards movement of the mouse will apply rotation about the player's X axis (pitch).

```
    // Update if we have moved
    if ( Direction > 0 || X != 0.0f || Y != 0.0f )
    {
        // Rotate our camera
        if ( X || Y )
        {
            // Are they holding the right mouse button ?
            if ( pKeyBuffer[ VK_RBUTTON ] & 0xF0 )
                m_Player.Rotate( Y, 0.0f, -X );
            else
                m_Player.Rotate( Y, X, 0.0f );

        } // End if any rotation
```

At this point we will have rotated the player. Now let us handle the movement. If any movement flags are set in the Direction variable then we need to apply some motor force to the player. We apply a motor force magnitude of 600 in this application but you can change this as you see fit. If the shift key is depressed, we set the applied motor force magnitude to 1000, which allows the shift key to enable a run mode for the player that moves it more quickly. We then call the CPlayer::ApplyForce function to add this motor force to the player.

```
        // Any Movement ?
        if ( Direction )
        {
            // Apply a force to the player.
            float fForce = 600.0f;
            if ( pKeyBuffer[ VK_SHIFT ] & 0xF0 ) fForce = 1000.0f;
            m_Player.ApplyForce( Direction, fForce );

        } // End if any movement

    } // End if camera moved
```

Remember that at this point the player has not had its position updated; we have simply added the force we have just calculated to its m_vecAppliedForce vector. The actual position change of the player happens in the CPlayer::Update method, which used to be called at the end of this function. This will

actually no longer be the case, since we have moved the CPlayer::Update call out of this function and into CGameApp::FrameAdvance just after the call to the CScene::AnimateObjects function.

We did this because, now that our scene geometry may be moving and the CPlayer::Update function is responsible for invoking the collision system and calculating the new position of the player, this must be called after the scene objects are updated. This will ensure that we are colliding with the position of these scene objects in their current state (instead of the previous frame state). In short, we should move the scene objects first and then try to move our CPlayer, and not the other way around. We will look at the CPlayer::Update method shortly.

The next section of code is new to the ProcessInput function and handles the reaction to the depression of the control key by the user. In this lab project, the control key will allow the player to jump up in the air. This is easily accomplished by getting the current velocity vector of the player and simply adding a value of 300 to its Y component. The code is shown below and uses two new functions exposed by CPlayer in this lab project.

```
    // Jump pressed?
    if ( m_Player.GetOnFloor() && pKeyBuffer[ VK_CONTROL ] & 0xF0 )
    {
        D3DXVECTOR3 Velocity = m_Player.GetVelocity();
        Velocity.y += 300.0f;
        m_Player.SetVelocity( Velocity );
        m_Player.SetOnFloor( false );

    } // End if Jumping Key
```

While our player is now allowed to jump, it must only be allowed to do so if its feet are on the ground. The control key should be ignored if the player is not in contact with the ground since there is no surface for their feet to push off from and thus no possible way to apply an upwards motor force. Later we will see how our traction coefficient allows for a little extra control when the player has jumped into the air.

In the above code you will notice that CPlayer now exposes a function called GetOnFloor that will return true if the player's feet are considered to be in contact with the ground. Do not worry about how this function works for the time being, as it is closely coupled with the collision detection update. For now, just know that if it returns true, our player is on the ground and should be allowed to jump. Notice that after we adjust the velocity of the player to launch our player into the air, we use the CPlayer::SetOnFloor function to inform the player object that its feet are no longer on the ground. This function will be discussed in a little while as well.

> **Note:** While you might assume that we would have modeled jumping as an upwards motor force, it turns out that it really is not worth the effort to go to so much trouble. Adjusting the velocity vector directly works just fine for our purposes and is certainly the simplest way to accomplish our objective.

Finally, if the 0 key on the num pad is pressed we adjust the player's camera offset vector to show the front view of the mesh (3rd person mode only).

```
    // Switch third person camera to front view if available.
    if ( m_pCamera && m_pCamera->GetCameraMode() == CCamera::MODE_THIRDPERSON )
    {
```

```
        if ( pKeyBuffer[VK_NUMPAD0] & 0xF0 )
            m_Player.SetCamOffset( D3DXVECTOR3( 0.0f, 26.0f, 55.0f ) );
        else
            m_Player.SetCamOffset( D3DXVECTOR3( 0.0f, 26.0f, -35.0f ) );

    } // End if
}
```

And there we have our new ProcessInput function. We have shown how it applies motor forces to the player object in response to user input. Again, note that at the end of this function, motor force has been applied to the player but no position update has been applied yet.

Introducing Resistance

We have now discussed what motor force is and how it is applied in the new CPlayer system. We have also mentioned that the amount of force we actually apply to the object to propel it forward is equal to the tractive force. The tractive force is the motor force scaled by the traction coefficient between the object and the surface (plus the addition of gravity as we will look at shortly).

But these are not the only forces we will have to apply to our object in order to calculate its new position in the CPlayer::Update function. If we were to simply add a constant tractive force to our player in each update call, our object would eventually accelerate to infinite speeds. Just think about it; if the velocity vector of the object is not diminished based on some concept like friction or drag, every time we applied a force, the velocity vector would get longer and longer. Since the (length of the) velocity vector is commonly referred to as the speed of the object, we know that our speed would increase with each update, forever. This makes perfect sense of course since we know from Newton's Second Law of Motion that

$$\mathbf{F} = m\mathbf{a}$$

If we rearrange this equation to look at the acceleration, we see that:

$$\mathbf{F}/m = \mathbf{a}$$

What this tells us is that our acceleration is always going to equal the total force applied to the object scaled by the inverse of the object's mass. Since the mass is a constant, you can see that if we assumed that mass equals 1 (for simplicity) our acceleration will equal our total applied forces. If we keep increasing our force, our acceleration will continue to increase. Since acceleration is simply the change in velocity with respect to time:

$$\mathbf{a} = \Delta\mathbf{v} / \Delta t$$

we know that over time, our velocity will simply get larger and larger (i.e., the speed increases because the object continues to accelerate).

Clearly this is not the case in real life. Just as we have tractive force propelling the object forward, we also experience forces between the surface and the moving object which counteracts the tractive force we are trying to apply to varying degrees. As we discussed earlier, these forces fall under the category of resistance forces (or contact forces for object in contact). We learned that on Earth, all solid bodies experience resistance to motion. Whether they are sliding on solid surfaces, rolling along, or moving through a liquid or gas, some amount of resistance is to be expected. We looked at multiple types of friction and also talked about viscous drag. Now let us look at how these ideas will apply in our player physics model.

Surface Friction – m_fSurfaceFriction

As mentioned earlier, friction forces depend on the types of surface that are in contact and we can generally break down the concept of friction into two categories: static friction and dynamic (or sliding) friction. In our demonstration, rather than deal with multiple friction models and how they influence forces, we are simply going to focus on two high level ideas – forces that speed us up (positive forces) and those that slow us down (negative forces). The motor forces and gravitational force fall into the category of positive forces because they will act to speed us up. Friction and drag are going to fall into the category of negative forces because they will serve to slow us down.

It is worth noting that we are not going to bother modeling static friction in our demonstration. Our goal is to slide our player on command based on user input and we decided not to worry about overcoming a friction threshold before we can begin movement. After all, while we may be sliding a sphere around the environment in our code, we are assuming our player is walking, not sliding around in the world. So we need to take some liberties with the theoretical models and focus on what works for us in this application. In a sense we are trying to model what is a nearly frictionless environment so that we can get smooth rapid movement, but be sure to include just enough dynamic friction to be able to slow us down (and maybe even do so differently depending on the surface material properties when desired).

To be fair, there is a downside to not modeling static friction which comes into play on sloped surfaces - due to the force of gravity, a motionless player on an inclined plane will slide downwards. This is not a terribly difficult problem to solve and there are a number of ways to tackle it should you decide you want static friction in your game. Keep in mind that should you decide to model actual static friction, calculating the normal force can be a little tricky. While you do get back the normals of the colliding triangles from the collision system, do not forget that you may have cases where you are intersecting more than one surface simultaneously. Averaging surface normals might be one way to tackle this, but in truth, the results are not very dependable. Rather than model pure static friction, you could simulate it far more cheaply by simply zeroing out your velocity vector (when you are not moving of course) when the surface normal is less than some particular inclination.

In this application we will stick to modeling dynamic friction only, since our primary movement model is based on sliding anyway. Our equation for dynamic friction was as follows:

$$\mathbf{F} = -\mu\, m\mathbf{g}$$

If we wanted to, we could plug in proper friction coefficients or even attempt to calculate our own based on the above formula. But for the most part, it fairly common to arrive at some fixed constant value(s) that produces a good feeling result in the game. Again, for simplicity in this lab project, rather than have every surface in our game store friction coefficients, we will store a single coefficient in our CPlayer class and apply it universally to all surfaces. Feel free to alter this behavior should you deem it necessary for your particular needs.

Our formula for calculating dynamic friction will be very straightforward. Friction will act in the opposite direction of our current movement vector, and our goal is to essentially have a force that will act to reduce the speed of the player over time. We can model idea this using the following calculation:

Friction = -m_fFriction * Velocity

> **Note:** Depending on how you decide to represent and store your friction coefficients, you could either calculate the constant as (1 – m_fFriction) or just m_fFriction.

In this case, Velocity is the current velocity vector of our player and m_fFriction is the friction coefficient. Again, we are using a single surface friction coefficient in our lab project but this could be extended and stored at the per polygon level.

Note that the friction calculation we are using is pretty much identical to the formula we learned for viscous drag for slow moving objects:

$$\mathbf{F}_R = -k\mathbf{v}$$

The resulting friction vector in the above calculation can later be added to our velocity vector. Since the sign of the friction coefficient is negative, this generates a vector pointing in the opposite direction of the velocity vector with the length scaled by the friction coefficient. Adding this vector to the velocity vector would essentially subtract some amount of length (i.e., reduce speed) from the velocity vector each frame, eventually whittling it down to zero magnitude over a number of frames (assuming no further motor forces were applied).

We can set the friction coefficient of our CPlayer object using the CPlayer::SetFriction function. This same friction coefficient is used for every surface, so there will be constant friction force working against our velocity vector in each update.

Air Resistance / Drag – m_fAirResistance

Another negative force we will consider is that of air resistance / drag. The amount of air resistance we will apply is going to be calculated using a drag coefficient just as we discussed earlier. This coefficient will ultimately describe how aerodynamic the object is. The less aerodynamic the shape of an object, the more air resistance will be experienced. Drag is an extremely important force to consider when modeling the movement of high speed objects, such as a car in a racing game.

The drag vector can be calculated with the following calculation where m_fAirResistance is a scalar value assigned to the object describing how aerodynamic it is (drag coefficient). Other factors could also

be considered when calculating this drag coefficient such as the type and density of the fluid being navigated, or even wind strength and direction (although this could be modeled as a separate motor force). In our demonstration we will actually use the higher speed version of our drag formula, but you can obviously experiment with this as you see fit. Recall that our formula was:

$$\mathbf{F}_R = -k\mathbf{v}^2$$

In code then, we will simply calculate the drag force as:

vecDragForce = -m_fAirResistance * m_vecVelocity * |m_vecVelocity|

Once again, we can see that this vector represents force acting in the exact opposite direction of the velocity vector. The length of this vector is equal to the squared length of the velocity vector scaled by our drag coefficient.

Gravity – m_vecGravity

As mentioned earlier, gravity is another force that we will include in our model. Keep in mind that gravity has been part of our CPlayer object since its first incarnation. We will represent gravity using a vector which will be directed down the -Y axis in world space. While it is not a system necessity that the gravity vector point down, this is obviously going to be preferable in most real world simulations. The magnitude of the gravity vector describes the strength of the gravitation pull of the planet on which our player is moving. To calculate the gravitational force applied to the object, we will turn to Newton's Second Law of Motion and multiply the gravity (acceleration) vector by the mass of the object.

vecGravForce = vecGravityVector*m_fMass

Updating our Velocity Vector

As discussed, we will apply motor force to our player object using the CGameApp::ApplyForce function (called from the CGameApp::ProcessInput). Regardless of how many times this function is called prior to the CPlayer::Update function, the forces will be combined in the m_vecAppliedForce vector.

After CGameApp::FrameAdvance has called the CGameApp::ProcessInput function to apply any forces, and has called the CScene::AnimateObjects function to update the scene and collision database, it next calls the CPlayer::Update function. Prior to going into this function we know that the m_vecAppliedForce member will contain a combination of all the motor forces applied in this iteration of the game loop.

As m_vecAppliedForce contains all motor forces we have applied to the object, we will first scale this vector by the traction coefficient to generate part of our final tractive force. This force vector will describe the actual force applied in the requested direction of motion. As mentioned, we will also include gravity in our tractive force. We will treat this a little differently from the motor force case and it

will not be scaled by our traction coefficient. Therefore, to generate the final tractive force we will scale the gravity vector by the mass of our player object and add the result to the applied force(s) vector.

m_vecAppliedForce = m_vecAppliedForce * m_fTraction
m_vecTractiveForce = m_vecAppliedForce + (m_vecGravity * m_fMass)

At this point, m_vecTractiveForce contains a vector describing the direction and force that should be applied to the object. Note that by giving ourselves a little bit of traction while we the player is airborne (we can set the value fairly low, but not quite 0), we are given a minor degree of control during jumps without having to write a lot of extra code. You can think of it as giving the player a bit of 'grip' on the air even though there is no actual surface to push off from.

Going back to our earlier discussion of what we are attempting to model at the high level, this vector represents our positive forces (i.e., those that will attempt to accelerate the player). Next we need to focus on generating our negative forces (i.e., the resistance forces acting against the force vector we have just created).

We start with the dynamic friction force. As discussed, this will be calculated as the negated result of the friction coefficient multiplied by the object's current velocity:

m_vecFrictionForce = - m_fSurfaceFriction * m_vecVelocity

While this approach would work fine, we decided to tweak the model a bit and take into account our traction idea as well. This could be described as a bit of a hack, but it makes it much easier to adjust the starting and stopping properties of the object given that we are using a constant friction in our demonstration. While the surface friction coefficient will be an arbitrary value, the traction coefficient will always be in the 0.0 to 1.0 range. By scaling the surface friction coefficient by the traction coefficient, we essentially make the assumption that if the friction of the surface is low (such as ice), it will take our player a while to ramp up to full speed. It would also take the player longer to stop given the difficulty getting a good grip on the surface. This friction force is going to be subtracted from the velocity vector of the player each time:

m_vecFrictionForce = - (m_fTraction * m_fSurfaceFriction) * m_vecVelocity

Next we calculate our final resistance force -- drag. Since we are decided to use the drag calculation for high speed movement, this force will be calculated by scaling the drag coefficient by a vector that is equal in direction to our velocity but with a squared length and then negating the result.

m_vecDragForce = - m_fAirResistance * (m_vecVelocity * | m_vecVelocity |)

We can now calculate the total force working on the body by summing the tractive force vector, the friction force vector, and the drag vector:

m_vecTotalForce = m_vecTractiveForce + m_vecFrictionForce + m_vecDragForce

Now that we know the total amount of force acting on the object, we use Newton's Second Law to do the rest. We divide out total force by the mass of the object to calculate the actual acceleration (or deceleration) of the object:

m_vecAcceleration = m_vecTotalForce / m_fMass

Now we must scale the acceleration by the elapsed time between frames so that we can determine the amount of acceleration to apply for this frame update. Recall that our definition of acceleration was a change in velocity with respect to time:

$$\mathbf{a} = \Delta\mathbf{v} / \Delta t$$

How can we use this knowledge to figure out what our new velocity needs to be? Well, for starters, let us consider what the change in velocity represents. If we had a previous velocity (v0) and we wanted to determine our new velocity (v1) based on our acceleration, we could say the following:

$$\Delta\mathbf{v} = \mathbf{v}1 - \mathbf{v}0$$

$$\mathbf{a} = (\mathbf{v}1 - \mathbf{v}0) / \Delta t$$

$$\mathbf{a}\Delta t = \mathbf{v}1 - \mathbf{v}0$$

$$\mathbf{v}0 + \mathbf{a}\Delta t = \mathbf{v}1$$

So we can see that all we need to do is scale our acceleration by the elapsed time, add it to our old velocity, and we will have our new velocity:

m_vecVelocity += m_vecAcceleration * TimeScale

Remember that the total force we calculated above may be acting primarily against the current velocity vector if the resistant forces are stronger than the sum of applied forces. This is what allows our player's velocity to decrease (i.e., deceleration) when the user lets go of the movement keys.

Our final important step will be resetting the applied forces vector to zero for the next frame since we have now used up all the force it contained updating the velocity vector.

We will grant that has been a very simplified introduction to physics, but it has given us all we need to implement fairly good handling in our CPlayer::Update function. You are strongly encouraged to take your knowledge further by trying out the Game Physics course, where you will get much more detail about the concepts discussed here and explore many other interesting topics in the world of physics.

We will now discuss the functions from the CPlayer class that have been updated.

CPlayer::CPlayer()

The constructor has had very few changes made to it. The friction, traction, drag coefficient, and the mass of the player object are set inside the constructor of CPlayer to default values. We use these default values in our application but you can change these values using the CPlayer methods as needed.

```
CPlayer::CPlayer()
{
    // Clear any required variables
    m_pCamera            = NULL;
    m_p3rdPersonObject   = NULL;
    m_CameraMode         = 0;
    m_nUpdatePlayerCount = 0;
    m_nUpdateCameraCount = 0;

    // Players position & orientation (independent of camera)
    m_vecPos             = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_vecRight           = D3DXVECTOR3( 1.0f, 0.0f, 0.0f );
    m_vecUp              = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
    m_vecLook            = D3DXVECTOR3( 0.0f, 0.0f, 1.0f );

    // Camera offset values (from the players origin)
    m_vecCamOffset       = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_fCameraLag         = 0.0f;

    // The following force related values are used in conjunction with Update
    m_vecVelocity        = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_vecGravity         = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_vecAppliedForce    = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_fTraction          = 1.0f;
    m_fAirResistance     = 0.001f;
    m_fSurfaceFriction   = 15.0f;
    m_fMass              = 3.0f;

    // Default volume information
    m_Volume.Min         = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
    m_Volume.Max         = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );

    // Used for determining if we are on the floor or not
    m_fOffFloorTime      = 1.0f;

    // Collision detection on by default
    m_bCollision         = true;
}
```

Notice that there is a new member variable called m_fOffFloorTime. This is used to track how long the player has not been making contact with the ground. It works as follows…

Every time the collision query is run on the player (shown in a moment) we will get back the bounding box describing the extents of the intersections that occurred between the player object and the environment. We can use this box to determine whether intersections have happened between the underside of the ellipsoid and a scene polygon. If so, we can assume that the player is in contact with the ground and the CPlayer::SetOnFloor method is called with a parameter of true.

```
void CPlayer::SetOnFloor( bool OnFloor )
{
    // Set whether or not were on the floor.
    if ( OnFloor )
        m_fOffFloorTime = 0.0f;
    else
        m_fOffFloorTime = 1.0f;
}
```

As you can see, this sets the value of the m_fOffFloorTime variable to zero. This happens every frame when the underside of our ellipsoid is found to be making contact with a surface.

Going further, in every subsequent frame (in the CPlayer::Update function), this variable is incremented by the elapsed time. Thus, if the ellipsoid is always making contact with the ground, it will be getting incremented by some small value and then ultimately reset to zero each frame by the collision test. However, if the collision query finds that the ellipsoid is not in contact with the ground, the SetOnFloor function is not called, but the value is still incremented so the off floor timer continues to increase.

We know that if the player is not making contact with the floor, they should not be allowed to jump. We also saw in the CGameApp::ProcessInput function that it used the CPlayer::GetOnFloor function to determine whether the player is considered to be making contact with the floor or not. It was only when this function returned true that a control key press was processed and the player's Y velocity adjusted to launch the player up into the air. The CPlayer::SetOnFloor function was then called to inform the player that it is currently in the air.

The CPlayer::GetOnFloor function is shown below.

```
bool CPlayer::GetOnFloor() const
{
    // Only return true if we've been off the floor for < 200ms
    return (m_fOffFloorTime < 0.200);
}
```

As you can see, it only returns true if the off floor timer has not been incremented past 200 milliseconds. Now, we know that in reality, as soon as this timer is not equal to zero, the player is off the ground and the function should return false for floor contact. The problem with doing it in that way is that it makes our system a little too sensitive. There may be times when our player is minimally bouncing along a surface and may be just a bit off the surface for a very small amount of time. In such cases we do not want to disable the jump ability since the player is not really supposed to be considered to be in the air; it may just be bumping over rough terrain for example. Therefore, only if the player has been off the ground for 200 milliseconds or more will we consider the character to be truly off of the ground and return false.

When we look at the constructor we are also reminded that our CPlayer object has two vectors describing its bounding box. The half length of this box will be used as the radius vector of the ellipsoid used by the collision system. That is, the bounding box describes a volume that completely and tightly encases the ellipsoid.

CGameApp::SetupGameState

The CGameApp::SetupGameState function of our framework is called to allow the application to initialize any objects before the game loop is started. This is where the player object has many of its properties set to their starting values. Some of the values shown here may vary from the actual source code in Lab Project 13.1 due to some last moment tweaking.

```
 void CGameApp::SetupGameState()
{
    // Generate an identity matrix
    D3DXMatrixIdentity( &m_mtxIdentity );

    // App is active
    m_bActive = true;

    m_Player.SetCameraMode( CCamera::MODE_FPS );
    m_pCamera = m_Player.GetCamera();
```

First we set the player object (a member of CGameApp) into first person mode. We then grab a pointer to the player's camera.

Next we set the acceleration due to gravity which will be applied to the player object during updates. Again, our gravity vector magnitude was basically determined by trial and error. Feel free to modify any of these values to better suit your own needs. We also set the offset vector describing the position of the player's camera relative to its position and set the camera lag initial to zero seconds.

```
    // Setup our players default details
    m_Player.SetGravity( D3DXVECTOR3( 0, -800.0f, 0 ) );
    m_Player.SetCamOffset( D3DXVECTOR3( -3.5f, 18.9f, 2.5f ) );
    m_Player.SetCamLag( 0.0f );
```

We then set up a suitable bounding box that encases the mesh of our CPlayer object. We use the CPlayer::SetVolumeInfo to set the player's bounding volume.

```
    // Set up the players collision volume info
    VOLUME_INFO Volume;
    Volume.Min = D3DXVECTOR3( -11,  -20, -11 );
    Volume.Max = D3DXVECTOR3(  11,   20,  11 );
    m_Player.SetVolumeInfo( Volume );
```

Next we set the camera's viewport properties and its bounding volume:

```
    // Setup our cameras view details
    m_pCamera->SetFOV( 80.0f );
    m_pCamera->SetViewport(   m_nViewX,
                              m_nViewY,
                              m_nViewWidth,
                              m_nViewHeight,
                              m_fNearPlane,
                              m_fFarPlane );
```

```
    // Set the camera volume info (matches player volume)
    m_pCamera->SetVolumeInfo( Volume );
```

Then we set the initial position of the player in the world.

```
    // Lets give a small initial rotation and set initial position
    m_Player.SetPosition( D3DXVECTOR3( 50.0f, 50.0f, 20.0f ) );

    // Collision detection on by default
    m_bNoClip = false;
}
```

CPlayer::Update

Before we look at the code to the CPlayer::Update function, you should be aware of a new line of code that has been added to the CScene::LoadSceneFromIWF function. This new line of code registers a CScene callback function with the player that will be called from the update function. We will see this in a moment.

Excerpt from CScene::LoadSceneFromIWF

```
GetGameApp()->GetPlayer()->AddPlayerCallback( CScene::UpdatePlayer, this );
```

With this knowledge in hand, we can look at the modified Update method which is now called from CGameApp::FrameAdvance just after the input has been processed and the scene objects have been animated. In this function, we will use the physics model we discussed earlier. CPlayer::Update is passed one parameter, the amount of time (in seconds) that has elapsed since the previous Update call.

The first thing we do is scale the applied motor forces by the traction coefficient to get the first part of our tractive force. We then add our gravitational force calculated by multiplying the gravity acceleration vector by the mass of the object:

```
void CPlayer::Update( float TimeScale )
{
    D3DXVECTOR3 vecTractive, vecDrag, vecFriction, vecForce, vecAccel;
    bool        bUpdated = false;
    float       fSpeed;
    ULONG       i;

    // Scale our traction force by the amount we have available.
    m_vecAppliedForce *= m_fTraction;

    // First calculate the tractive force of the body
    vecTractive = m_vecAppliedForce + (m_vecGravity * m_fMass);
```

At this point we have our total tractive force. Now it is time to calculate the resistance forces that act against this tractive force vector.

First we will calculate drag based on our model discussed earlier (for high speed objects):

```
    // Now calculate the speed the body is currently moving
    fSpeed = D3DXVec3Length( &m_vecVelocity );

    // Calculate drag / air resistance (relative to the speed squared).
    vecDrag = -m_fAirResistance * (m_vecVelocity * fSpeed);
```

Next we calculate the dynamic friction vector, once again using the model introduced earlier (similar to viscous drag for slow moving objects). We can then add the friction vector, drag vector, and the tractive force vector together and divide the resulting vector to get our total force. Dividing by the mass of the object gives us the current acceleration of the object.

```
    // Calculate the friction force
    vecFriction = -(m_fTraction * m_fSurfaceFriction) * m_vecVelocity;

    // Calculate our final force vector
    vecForce = vecTractive + vecDrag + vecFriction;

    // Now calculate acceleration
    vecAccel = vecForce / m_fMass;
```

Now we can scale our acceleration by the elapsed time to get an acceleration value for this frame update. The result is then added to our previous velocity vector and we have our final velocity for this frame. The applied forces vector is then reset to zero since we have used up all applied forces at this point.

```
    // Finally apply the acceleration for this frame
    m_vecVelocity += vecAccel * TimeScale;

    // Reset our motor force.
    m_vecAppliedForce = D3DXVECTOR3( 0.0f, 0.0f, 0.0f );
```

We have now successfully calculated our new velocity vector.

In the next step we will loop through any callback functions that have been registered for the player. In our application there will be just one -- a pointer to the CScene::UpdatePlayer function which is responsible for running the collision query. We will see this in a moment when we discuss the source code.

```
    // Only allow sources to fix position if collision detection is enabled
    if ( m_bCollision && m_nUpdatePlayerCount > 0 )
    {
        // Allow all our registered callbacks to update the player position
        for ( i = 0; i < m_nUpdatePlayerCount; i++ )
        {
            UPDATEPLAYER UpdatePlayer =(UPDATEPLAYER)m_pUpdatePlayer[i].pFunction;

            if ( UpdatePlayer( m_pUpdatePlayer[i].pContext, this, TimeScale ) )
                bUpdated = true;
        }
    } // End if collision enabled
```

At this point the callback function would have been called to run a collision query on the position and velocity of the player. It would have now set the player in its new non-colliding position. We will look at the callback function next since it is the final piece in our puzzle.

In the next section of code we call some legacy code if the collision system is not being invoked or if no callback functions have been registered to update the position of the player. When this is the case the pUpdated member will not have been set to true (see above code), so we scale the new velocity vector we just calculated by the time scale and pass this movement vector into the CPlayer::Move function to physically alter the position of the player. This next section of code will not be executed by our application in Lab Project 13.1 since we are using the collision system.

```
    if ( !bUpdated )
    {
        // Just move
        Move( m_vecVelocity * TimeScale );

    } // End if collision disabled
```

Next we call the Update method of the player's camera object so that it can also alter its position. This is important when the camera is a third person mode since it will cache the new position of the player and start to track it (with the desired amount of lag). This function is a no-op for other camera modes since they do not implement this virtual function.

```
    // Let our camera update if required
    m_pCamera->Update( TimeScale, m_fCameraLag );
```

If this is a no-op for other camera modes, you may be wondering how the position of those cameras are updated. The answer will become clear when we look at the callback function momentarily.

Next we also loop through any callback functions which may have been registered for the player's camera and call those functions as well. This allows us to register functions which will handle the collision of the camera with the scene if we wish. If this callback function calling code looks unfamiliar to you, refer back to Chapter 4 where we first implemented and discussed this callback system.

```
    // Only allow sources to fix position if collision detection is enabled
    if ( m_bCollision && m_nUpdateCameraCount > 0 )
    {
        // Allow all our registered callbacks to update the camera position
        for ( i =0; i < m_nUpdateCameraCount; i++ )
        {
            UPDATECAMERA UpdateCamera =(UPDATECAMERA)m_pUpdateCamera[i].pFunction;

            UpdateCamera( m_pUpdateCamera[i].pContext, m_pCamera, TimeScale );

        } // Next Camera Callback

    } // End if collision enabled.
```

Next we use the GetOnFloor function to adjust the traction and surface friction coefficients. If the player is not currently in contact with the floor, we will assume that there is no surface friction acting against its velocity. It would seem strange if the velocity of our player was being diminished by surface friction if the player is currently falling through the air.

As mentioned earlier, in reality the player should not have any traction in the air since there is nothing for the player's feet to push against. In this case we will set the surface friction coefficient to zero but set the traction coefficient to a very low value. Although it would be more correct to set the traction to zero too, we set it to a very low value so that the player still has limited directional control even when in mid air. This is consistent with gameplay mechanics in many first and third person titles. If the player is in contact with the floor, we set its traction and friction values to 1.0 and 10.0 respectively. Feel free to experiment with these values to change the feel of the player as it navigates the scene to something that suits your tastes.

```
    if ( !GetOnFloor() )
    {
        SetTraction( 0.1f );
        SetSurfaceFriction( 0.0f );

    } // End if not on floor
    else
    {
        SetTraction( 1.0f );
        SetSurfaceFriction( 10.0f );

    } // End if on floor
```

Finally, we increment the m_fOffFloorTimer value by the elapsed time as we must do each frame. We will see in a moment how this value is set to zero when the collision test determines that the player is in contact with the ground. Only when this variable has been incremented past 200 (milliseconds) do we consider the player to truly be off the ground for a significant enough amount of time for us to consider the player to be in the air. Thus it is only when this variable is greater than or equal to 200 does the GetOnFloor function returns false.

```
    // Increment timer
    m_fOffFloorTime += TimeScale;

    // Allow player to update its action.
    CPlayer->ActionUpdate( TimeScale );
}
```

As the final line of the function you can see us call the CPlayer::ActionUpdate function, which was added in the previous lesson. This function adjusts the current animation action of the third person object attached to the player. It sets the object's world matrix and also sets the actor action based on whether the character is idle or shooting, for example. Any other animations you wish to be played based on input or game events should be placed in this function.

We have now seen how the Update function of the player has been enhanced to include a more realistic physics model. We have not yet seen where the velocity vector we calculate in the above code is

actually used to update the position of the player object. As discussed, this is done in the CScene callback function that is registered with the player and called from the function previously discussed. Let us look at this callback function now to conclude our discussion.

CScene::UpdatePlayer

This callback function is the glue that holds this whole application together. It is called from the previous function and is passed a pointer to the player we wish to move. It is also passed the elapsed time between frames (in seconds) so that it knows how to scale that movement. As the code to the previous function demonstrated, when this function is called, the current velocity vector has been calculated using our player physics model so that at this point, the player object contains a velocity vector describing the direction and speed it would *like* to move. This function will call the collision system to see if the player can be moved along the velocity vector without obstruction, and if not, the collision system will return a new position for the player that is free from intersection. This function will then update the position of the player as dictated by the collision system. The collision system also returns a new velocity vector describing the direction and speed the player should be traveling after intersections have been factored in. We will see in a moment that we will not simply set the velocity of the player to the one returned from the collision system since this will cause some problems in our physics model. Instead, we will make a few adjustments so that our player's velocity will not diminish too significantly when trying to ascend shallow slopes.

We decided against placing the call to CCollision::CollideEllipsoid directly inside the CPlayer update function because then the CPlayer object would be reliant on the CCollision class and the design of CPlayer would be less modular. It makes sense that the callback function used for this purpose should be part of the CScene namespace since it is the scene object that contains both the scene geometry and the collision database (CCollision). The scene object also has the functions that load the scene and register it with the collision database. So it seemed a sensible design that this object should also be the one responsible for running queries on the collision database in our application.

Let us look at this function a little bit at a time. The function (like all player callback functions) accepts three parameters. The first is a void context pointer that was registered with the callback function. This data pointer can be used to point at anything, but we use it when we register the callback function to store a pointer to the CScene object. This is important because in order for the CScene::UpdatePlayer function to be a callback function, it must be static and therefore have no automatic access to the non-static members of the CScene class. In our code, this first parameter will point to the instance of the CScene object that contains the collision geometry. For the second parameter, a pointer to the calling CPlayer is passed. The third parameter will be the elapsed time since the last update.

```
bool CScene::UpdatePlayer( LPVOID pContext, CPlayer * pPlayer, float TimeScale )
{
    // Validate Parameters
    if ( !pContext || !pPlayer ) return false;

    // Retrieve values
    CScene     *pScene   = (CScene*)pContext;
    VOLUME_INFO Volume   = pPlayer->GetVolumeInfo();
    D3DXVECTOR3 Position = pPlayer->GetPosition();
```

```
    D3DXVECTOR3 Velocity = pPlayer->GetVelocity();
    D3DXVECTOR3 AddedMomentum;

    D3DXVECTOR3 CollExtentsMin, CollExtentsMax, NewPos, NewVelocity;
    D3DXVECTOR3 Radius = (Volume.Max - Volume.Min) * 0.5f;
```

In the first section of code we cast the context pointer to an object of type CScene so we can access the scene's methods and member variables. We also extract the position, velocity and bounding box information from the player into local variables. The ellipsoid that will be used to approximate the player in the collision system has its radius vector generated by using half the length of the bounding box. Remembering that the radius vector describes the distance of the ellipsoid surface from the center of the ellipsoid, we can see that this describes an ellipsoid that fits relatively tightly in the player's bounding box. We also allocate two 3D vectors called CollExtentsMax and CollExtentsMin which will be used to transport information about the extents of the collisions that occur with the ellipsoid during the detection process.

We now have all the information we need to run the collision test.

```
    // Test for collision against the scene
    if ( pScene->m_Collision.CollideEllipsoid(  Position,
                                                Radius,
                                                Velocity * TimeScale,
                                                NewPos,
                                                NewVelocity,
                                                CollExtentsMin,
                                                CollExtentsMax ) )
    {
```

When the CollideEllipsoid function returns, the local variables NewPos and NewVelocity will contain the new position and velocity of the object calculated by the collision system. The CollExtentsMax and CollExtentsMin vectors will describe the maximum and minimum points of intersection along the X, Y and Z axes that occurred in this movement update.

The first thing we do is test the minimum Y coordinate of the intersections bounding box returned by the collision system. If the minimum point of intersection along the Y axis is smaller than the Y radius of the ellipsoid, minus a quarter of the Y radius of the ellipsoid, then it means we have an intersection that occurred with the ellipsoid in its bottom quarter (roughly) as shown in Figure 13.11.

Figure 13.11

When this is the case, we consider the ellipsoid to be in contact with the floor and we call the CPlayer::SetOnFloor function to reset the player's m_fOffFloorTimer to zero. It will take at least another 200 milliseconds of non-contact between the ellipsoid and the floor for the player to be considered in the air.

```
        // If the lowest intersection point was somewhere in the
        // bottom quarter (+tolerance) of the ellipsoid
        if ( CollExtentsMin.y < -(Radius.y - (Radius.y / 4.25f)) )
        {
            // Notify to the player that were in contact with a "floor"
            pPlayer->SetOnFloor( );

        } // End if hit "feet"
```

The reasons for the existence of the next section of code will require some explanation as it is responsible for generating the new velocity vector of the player. Is it not true that the collision function returned us the new integration velocity? Well, yes and no.

It is true that the collision system does correctly calculate our new velocity, and in a perfect world we could simply set this new velocity as the player's velocity. If collisions occurred, the velocity vector will now be pointing in the slide direction, which is what allows our collision system to slide our player over bumps and steps. If we were passing in a really large initial velocity vector each time, we would cruise over most slopes and bumps with little trouble. However, now that we have integrated more complex physics into our model, our velocity also has resistant forces working against which diminish it each frame. Therefore, the amount of motor force we would usually apply to allow the ellipsoid to slide around a section of flat floor will not be enough to slide the same ellipsoid up a steep slope.

If we think about this in real terms, imagine applying enough force to a ball so that it slides at a fairly slow constant speed across your carpet. Now imagine that the ball hits a step. If you applied the same force to the ball, it would be nowhere near enough to allow you to push the ball up and over the step. The same is true here. What we consider a nice motor force for the player when moving along flat ground will not be enough to slide up most ramps with gravity and drag working against us.

We know that our collision system will project the input velocity vector onto the sliding plane so that the direction of the player changes to slide along corners and up steps. However, we also know that if our ellipsoid was to hit a step of significant size, the velocity vector would have its direction diverted so that it pointed up into the air, for the most part. The velocity vector returned will have most of its movement diverted from horizontal movement along the X and Y axes to vertical movement along the Y axis. Even if the player had enough force so that the projected velocity vector was enough to push the ellipsoid up higher than the step, we will have lost most our forward (X,Y) momentum contained in the original velocity vector. Essentially, we might have enough projected velocity to clear the step vertically, but then we would not have enough to actually move forward and onto the step.

To solve this particular problem we could use only the Y component of the velocity vector returned from the collision system and use the X and Z components of the original velocity vector. This way, the collision system can correctly inform us about how we need to move up and down when a step is hit, but the original X, Z velocity is still maintained (not diminished) so we have enough horizontal force to move not only up the step, but up and over. It is vitally important that we always obey the Y component of the velocity vector returned from the collision system since this is what prevents us from falling through floors. Remember, we might pass in a velocity vector with a massive downwards force (e.g., a very strong gravitational force has acted) of say (0, 300, 0). The collision system would detect that we cannot fall downwards as described by the velocity vector because there is geometry underneath us and

the returned velocity vector would have a Y component of zero. So, the Y component of our player's velocity must be set to the exact Y velocity returned from the collision system.

Ignoring the returned X and Z velocity components and simply using the original (pre-collision) X and Z velocity components along with the Y velocity returned from the collision system would certainly solve our problem in one respect. It would mean that the collision system can tell us the new up/down velocity that should be applied to our position to clear obstacles and steep slopes, while the original horizontal velocity would keep us moving in the XZ direction with the original forces applied. This would allow us to glide over steps and slopes with ease since our horizontal momentum would not be diminished at all by the obstacles we encounter. Indeed we would be able to slide up very steep slopes; just about any slope that was not perfectly vertical. Of course, that in itself is problematic since we do not want our player to be able to slide up near perfect vertical slopes.

To be sure, ignoring the X and Z components is certainly what we want to do to help us get over small obstacles. That is, we want to obey their direction but not necessarily the length. We need a way of knowing that if the object is fairly small, like a shallow slope or a small step, then we should use the original (pre-collision) X and Z velocity components so that we slide up **and over** the obstacle with ease. However, when we are dealing with larger obstacles or much steeper slopes, we do not want to ignore the X and Z velocity components returned from the collision system. In this case, they describe the way our horizontal momentum should be diminished, preventing us from climbing very steep slopes or large steps. Doing otherwise would seem unnatural unless you were making a Spiderman® style game and it is your intention to allow the player to traverse vertical walls.

So it seems that the degree to which we ignore the diminished X and Z components of the velocity vector returned by the collision system depends on how tall/steep the object we have collided with is. Fortunately, this is no problem; we can determine how high the object we collided with was because we have the collision extents bounding box. We will use the maximum height of the intersection to scale the amount that the horizontal velocity was diminished by the collision system.

Let us get started and see how this will work.

First, the velocity vector returned from the collision system is a per-frame velocity, but when performing our physics calculations we work with per-second values. By dividing the returned velocity vector by the elapsed time we get the new velocity vector specified as a per-second velocity. This is the velocity vector returned from the collision system describing the new direction and speed we should be traveling. We then copy the Y component of the velocity vector returned by the collision system into the velocity vector of our player object. We must retain the Y component of the collision velocity vector because it keeps us from falling through the floor.

```
        // Scale new frame based velocity into per-second
        NewVelocity /= TimeScale;

        // Store our new vertical velocity. Its important to ignore
        // the x / z velocity changes to allow us to bump over objects
        Velocity.y = NewVelocity.y;
```

We have now modified the Y velocity in accordance with the collision system. Next we will determine the maximum height of intersection and use that to scale the difference between our original undiminished horizontal velocity and the horizontal velocity returned from the collision system. It should be noted that this works well with slopes and not just steps, as shown in Figures 13.12 and 13.13

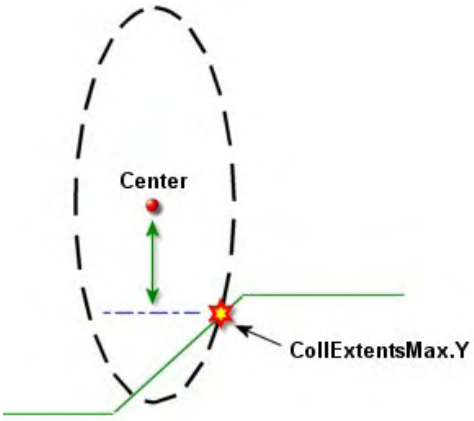


Figure 13.12

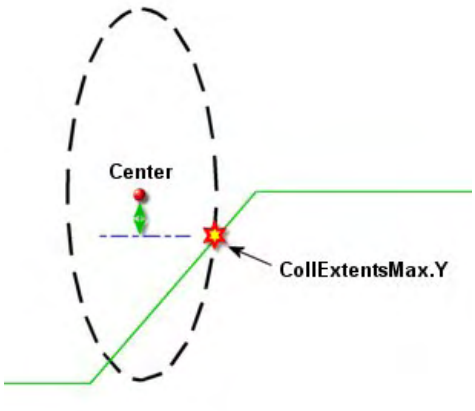


Figure 13.13

In Figure 13.12 we see an ellipsoid colliding with a fairly shallow slope and we can see that the point of intersection is further down the ellipsoid than the intersection shown in Figure 13.13 (where a taller, steeper slope is collided with). It is easy to see that the steeper the slope, the closer to being vertically aligned with the center point of the ellipsoid the intersection point will be. Therefore, we can scale the amount we want to diminish the horizontal movement of the player (as described by the collision system) based on how near to the center of the ellipsoid (vertically) the closest intersection point is. We can imagine for example, that if the ellipsoid collided with a brick wall, the intersection point would be perfectly aligned with the center of the ellipsoid and we should fully obey the collision system when it tells us we need to stop our horizontal movement immediately. On the other hand, if we find that the intersection point is closer to the bottom or top of the ellipsoid, we can ignore the new X and Z diminished velocity vector components returned by the collision system and use the original XZ velocity. This will allow us to slide up and over small obstacles or shallow slopes with ease. This works the same for collisions with both the top and bottom of the ellipsoid. We are interested in finding the intersection point that is closest to the center of the ellipsoid vertically since it is this ratio that will be used to scale the amount by which we factor in the diminished XZ velocity returned by the collision system. The next section of code will show how the new X and Z velocity components of the player's velocity vector are calculated.

First we will subtract about one quarter of the radius height from the radius. We are essentially going to say that if the closest intersection point to the vertical center of the ellipsoid is contained in the bottom quarter or top quarter of the ellipsoid's radius, this is a small obstruction. In that case we remove it completely and do not diminish our original horizontal momentum at all.

```
        // Truncate the impact extents so the interpolation below begins
        // and ends at the above the players "feet"
        Radius.y -= (Radius.y / 4.25f);
```

Having shaved a little over a quarter of the radius length (both sides of the center point), we will now find the Y coordinate of intersection that is closest to the center point. We will first take the maximum of the –Radius.y (the bottom of the new shaved ellipsoid) and the maximum Y intersection point. We will then take the minimum of the positive radius (top of the ellipsoid) and the maximum we just calculated.

```
        CollExtentsMax.y = max( -Radius.y, CollExtentsMax.y );
        CollExtentsMax.y = min( Radius.y, CollExtentsMax.y );
```

What we have now is the Y component of the intersection point that is closest to the center of the ellipsoid, or a value that is equal to the Y radius of the ellipsoid. We do this just to make sure that we get an intersection point along the diameter of the ellipsoid (-Radius to +Radius). This is important because if our player is falling though the air, it will not be colliding with anything and we need the value to be in the [–Radius, +Radius] range to perform the next section of code. Furthermore, because we shaved ¼ of the Y radius of the ellipsoid, if the maximum intersection point is in the bottom or top quadrant of the ellipsoid, it would initially be smaller than the shaved radius. That is why we clamp the values in the range of the shaved vertical diameter of the ellipsoid.

Now we will create a weighting value called fScale which will be calculated by dividing the absolute value of our closest Y point (closest to the ellipsoid center), by the radius of the ellipsoid and subtracting the result from 1.

```
        float fScale = 1 - (fabsf(CollExtentsMax.y) / Radius.y);
```

fScale is a parametric value in the range [0, 1] describing the Y difference between the ellipsoid center and our closest intersection point. If the intersection point is equal to the center of the ellipsoid (vertically), a value of 1.0 will be generated. If the closest intersection point is right at the top or bottom of the ellipsoid (or anywhere in the top or bottom zones), a value of 0.0 will be generated. We will now use this value to scale the difference between our original horizontal velocity (pre-collision) and the diminished horizontal velocity returned from the collision system.

```
        Velocity.x = Velocity.x + (((NewVelocity.x ) - Velocity.x) * fScale);
        Velocity.z = Velocity.z + (((NewVelocity.z ) - Velocity.z) * fScale);
```

As you can see, we subtract the original components from the new diminished horizontal velocity components:

(NewVelocity – Velocity)

Since NewVelocity is always smaller in the case of an intersection, this generates a vector acting in the opposite direction to that of the original velocity vector. Therefore, adding it to the original velocity vector actually subtracts the correct amount from its length such that it is equal to the diminished velocity vector. For example, if our original velocity vector was <10, 10> but the collision system returned a diminished velocity vector of <2, 2>, subtracting and adding to the original velocity vector would give:

OriginalVelocity += NewVelocity – OriginalVelocity
= <2, 2> <-10, -10>
= <-8, -8>

Therefore:

<10,10> += <-8,-8>
= <2 , 2>

That seemed like a lot of manipulation to do just to scale the original velocity so that it is equal to the velocity returned from the collision system. Why not just use the velocity returned from the collision system as the new velocity in the first place? Well, if that is all we intended to do then we would, but notice that the negative vector created from (NewVelocity – Velocity) is actually scaled by fScale. Therefore, the amount we diminish our velocity vector as dictated by the collision system, ends up being dependant on how close to the center of the ellipsoid a collision happened. As such, we have achieved our goal! For little bumps or shallow slopes, fScale will be close to zero, allowing us to continue our horizontal movement unhindered. When the slope is steep or the obstacle is tall, fScale will be closer to 1.0 and the full diminishment of horizontal momentum is applied to the player's velocity as directed by the collision system.

With our velocity vector now correctly calculated and the new position of the ellipsoid returned from the collision system, we can set them as the CPlayer's current position and velocity before returning from the function.

```
        // Update our position and velocity to the new one
        pPlayer->SetPosition( NewPos );
        pPlayer->SetVelocity( Velocity );

        // Hit has been registered
        return true;

    } // End if collision

    // No hit registered
    return false;
}
```

We have now covered the function in its entirety and have no further code to discuss. Do not expect to understand all of the code we have added in this lab project right away. Expect to have to study it again before you feel comfortable implementing such a system yourself from the ground up. The main areas of focus in this lesson have been the CCollision class and the updated CPlayer code.

Conclusion

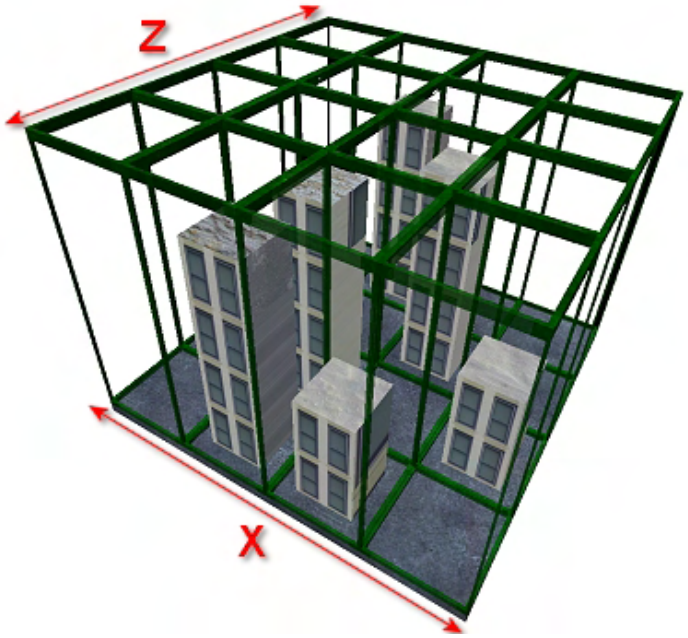
In many ways, this lesson has been a very exciting one. We now have a reusable collision system that can be plugged into our future applications to provide collide and slide behavior with arbitrary scene geometry. We have implemented it in such a way that our existing classes such as CActor can also be registered with the collision system's geometry database. We have even employed a system that allows us to support animating collision geometry when the situation calls for it.

We also upgraded our CPlayer object with a nicer (but still admittedly simplistic) physics model which allows us much greater control over how the player interacts with the collision environment. The CCollision class will be used in all lab projects from this point onwards.

In the next lab project we will implement hierarchical spatial partitioning to improve our collision system's performance (among other things). We will add a spatial partitioning system to our CCollision class so that a fast broad phase can be introduced to quickly identify potential colliders, even when dealing with huge data sets. The broad phase component will significantly speed up our collision system by performing efficient bulk rejection of polygons that are not contained within the same region of the scene as the ellipsoid that is currently being queried. This is going to be very important since our collision system currently has to perform the full range of intersection tests on every triangle that has been registered. This will simply not be good enough as we progress with our studies and start to develop larger and more complex scenes. Before moving on, please be sure that you are comfortable with the material covered in the textbook and workbook as we will see a lot of it again in the next chapter.

Chapter Fourteen

Hierarchical Spatial Partitioning



Introduction

In this lesson we will study various hierarchical spatial partitioning techniques and use them to implement a broad phase for the collision detection and response system developed in the previous chapter. The techniques we learn in this lesson and the next will also be used to speed up the rejection of non-visible polygon data from the rendering pipeline.

The concepts you learn in this chapter are some of the most important you will ever learn as a game developer, or even as a general programmer. Indeed you will find yourself using them at many points throughout your programming career to optimize a particular task your application has to perform. Although we will apply the tree traversal techniques discussed in this lesson to the area of 3D graphics programming, the usefulness of the ideas we will introduce reach far beyond game creation. For example, such techniques are used by image processing routines to optimize the mapping of true color images to a limited palette of colors. Hierarchical tree structures are also used to perform quick word searches in databases and efficiently sort values into an ascending or descending order. This lesson will be the first of four studying spatial partitioning. It will eventually lead to the implementation of a broad phase for our collision system and an efficient hardware friendly PVS rendering system.

Fortunately, many of the concepts we discuss will not require us to cover a lot of new ground since we already have a fair understanding of the two areas that hierarchical spatial partitioning essentially borrow from: parent/child relationships and bounding volumes. Combined, they allow us to spatially divide our scene into a hierarchically ordered set of bounding volumes that contain all of the scene geometry. These spatial trees can be quickly traversed and entire branches of the tree (along with child bounding volumes and the polygon/mesh data they contain) can be rejected from consideration with only a few simple bounding volume intersection tests.

Over the next few chapters we will be implementing spatial hierarchies in a variety of different flavors. We will implement a base tree class that can be used by the application to interface with all our spatial tree types. We will discuss and implement quad-trees, oct-trees, kD-trees and the famous (or infamous) BSP tree. These four tree types all provide a different way for the scene to be spatially subdivided and each has its uses in certain situations. The BSP tree (Chapters 16 and 17) will be vitally important as it will be used to aid us in the calculation of a Potential Visibility Set (PVS); this is a process that involves pre-computing what polygons can be seen from any point within the level at development time. The calculation of a PVS will allow our application to render scenes of an almost unlimited size (memory permitting) while keeping our frame rates fast and interactive. But before we get ahead of ourselves, let us focus on the basics, since there is a lot to cover.

This chapter will be focused on examining some of the common tree types that are used in games and the theory behind how they partition space and how they can be used. Later in this lesson and in our lab project, we will implement a broad phase for our collision system that will significantly speed up polygon queries on the collision system's geometry database. We will also discuss the usefulness of such systems for rendering, although this will not be implemented or demonstrated until the following lesson where a hardware friendly system will be introduced. Furthermore, in this chapter we will discuss the various utility techniques which typically accompany spatial partitioning algorithms: polygon clipping and T-junction repair, for example. So why don't we get started?

14.1 Introducing Spatial Partitioning

It is difficult to overstate the importance of spatial partitioning in today's computer games. Every game you currently play will undoubtedly use spatial partitioning as the core of its collision system and rendering logic. We learned in Chapter 4 of Module I how the frustum culling of a mesh's bounding volume could be used to prevent its polygon data from being passed through the transformation pipeline unnecessarily. Although the DirectX pipeline does perform a form of frustum rejection on our behalf, it does so at the per-polygon level, and worse still, only after the vertices of that polygon have undergone the calculations needed to transform them into the homogenous clip space.

We discovered that by surrounding a mesh with an axis aligned bounding box, we could test this box against the frustum to introduce a broad phase rejection pass in our mesh rendering logic. If we determined that the mesh's bounding box was outside the frustum, then we knew we did not have to bother passing its polygon data through the transformation pipeline. If that mesh was comprised of 20,000 polygons, we just avoided needlessly transforming them at the cost of a simple AABB/Frustum test. Of course, the same logic can be applied when performing any query on the scene that is required at the per-polygon level, such as we find with collision detection. We know that if our swept sphere does not intersect the bounding volume of a mesh then we do not have to test any of its individual triangles. We might perceive this to be a very simple broad phase implementation when added to our collision system. However, we do not always have our scenes comprised of multiple meshes, such as the internal geometry loaded from an IWF file for example. In such cases the entire scene is represented as a single mesh, so surrounding this with a single AABB would not help at all. That is, it would not allow us to reject portions of that mesh during frustum tests or collision queries. Spatial partitioning is the next level in bounding volume rejection and it works just as well for polygon soups as it does for scenes comprised of individual meshes.

In the first section of this lesson we will bias the demonstrations of spatial partitioning towards performing pure collision queries from the perspective of game physics, such as those discussed in the last chapter. But later we will discuss and implement a rendering system that will also benefit from spatial partitioning while remaining hardware friendly. We will also slant our early discussions toward the spatial partitioning of a polygon soup, although later we will discuss how spatial trees can also be built that partition complete meshes. In fact, the spatial trees we implement will be capable of managing both. That is, we can divide space into bounding volumes that contain both a list of polygons and a list of mesh objects that exist in that area of space. This allows our system to handle cases where the core geometry might be comprised of a huge list of static polygons but the dynamic objects are individual meshes or actors. We want both types to be supported by our tree so that we can quickly reject not only the core scene geometry that lay outside the frustum for example, but any dynamic objects (such as skinned characters) which are also not visible.

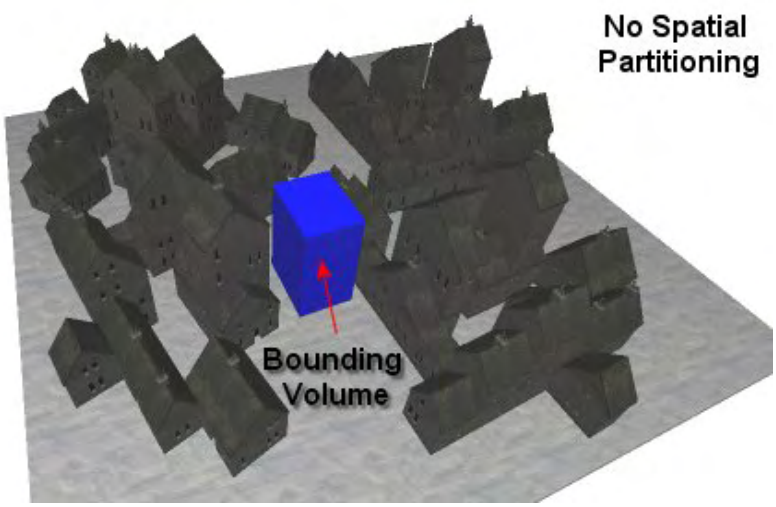


Figure 14.1

Let us begin our journey by imagining a simple town scene (Figure 14.1) consisting of 100,000 polygons. Assume that the blue bounding box in the figure represents an AABB that has been compiled around the source and destination positions that our swept sphere will move between in the current update. We used a similar technique when collecting the terrain polygons for our collision system in the previous chapter. You will recall that rather than trying to test the awkward shape of the swept sphere against the terrain to get a list of potential colliders, we simply built an AABB around the swept sphere and collected any terrain geometry that fell inside it. We will assume that the same logic is being applied here. After all, we are only interested in quickly finding which polygons will possibly intersect our swept sphere so that they can be passed on to the narrow phase. Only polygons that are contained partially or completely inside the bounding volume will need to be tested in the computationally expensive narrow phase of our collision system.

A naïve approach to a broad phase implementation might be to test every polygon in the scene against the AABB surrounding the swept sphere and reject all polygons that do not intersect it. This would certainly work and at the end of the process we would have compiled a list of polygons which intersect the AABB and thus have the potential to be a collider with the swept sphere. This list of potential colliders would then be passed to the narrow phase where the swept sphere intersection tests are performed on each polygon. Remembering that this scene is assumed to be comprised of 100,000 polygons, the pseudo-code to such a broad phase implementation would be as shown below.

> **Note:** We will assume for now that we have a function called 'AABBIntersectPolygon' which determines whether a polygon intersects an axis aligned bounding box. The function in this example is passed the minimum and maximum extents of the axis aligned bounding box and a pointer to the polygon that is to be tested.

```
for (i = 0; i < 100000; i++)                              // Loop through every poly
{
    CPolygon *Poly = CScenePolygonList[i];                // Get current poly to test

    if (AABBIntersectPolygon( BoxMin , BoxMax , CPoly ))  // Does it intersect AABB
    {
         PotentialColliderList.Add ( CPoly );               // Add to List that will
    }                                                       // be passed to narrow
}                                                           // phase

NarrowPhase ( PotentialColliderList );
```

In this example CPolygon is assumed to be the structure used to contain an individual polygon and CScenePolygonList is assumed to be an array that contains the list of polygons comprising the scene. BoxMin and BoxMax are the minimum and maximum extents of an AABB that encompasses the swept sphere's movement in this update. PotentialColliderList is some type of container class that manages a list of polygons and exposes methods allowing us to add polygons to its internal list. Whenever a polygon is found to be intersecting or totally inside the bounding box, it is added to this list. At the end of the loop the broad phase is complete and we have compiled a list of potential colliders. This list is then passed to the narrow phase where each polygon would have to be inspected against the swept sphere as described in the previous chapter.

This is a broad phase implementation to some degree, however any broad phase that requires 100,000 bounding volume tests just to find the potential colliders is obviously not very efficient. When we consider the incredible geometric detail in today's games, testing every individual polygon (even against a simple AABB) every time we wish to perform a query on the scene data is not going to suffice when it comes to performance. The idea of the broad phase is to quickly reject large blocks of polygon data from consideration very quickly. The number of tests needed to find the potential colliders in the broad phase should not come anywhere close the polygon count of the scene as is the case in the above example implementation. We need to be able to say, "My bounding volume is not located within this entire area, so all polygons in this area should be dismissed right away".

So let us start with a naïve approach to spatial partitioning, but one that will serve to solidify certain concepts in less obfuscated way before moving on to the subject of more common spatial hierarchies.

Let us imagine the same scene again, only this time, when the level was first loaded, we built an axis aligned bounding box around the entire level. Let us also imagine that with this information in hand, we divided the area of the scene's bounding box into a 7x7 grid (an example) of bounding boxes as shown in Figure 14.2.

Generating these 49 bounding boxes would be trivial. Once we had the bounding volume of the scene we could just divide its width and depth by 7 giving us a value of N and M for the width and depth deltas, respectively. We would then set up a loop to step through along the width of the scene's bounding box in steps of N and along the depth of the scene's bounding box in steps of M. In the inner loop (M), the coordinate (N*Column, M*Row) describes the minimum extent of the current bounding box being calculated and vector (N*Column + N, M*Row + M) describes its maximum extent.

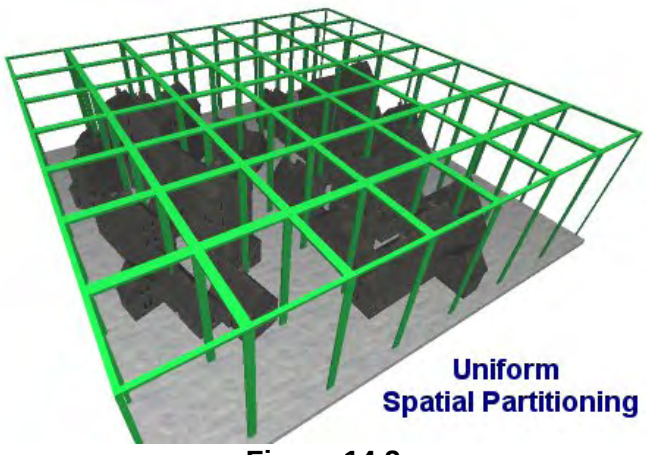


Figure 14.2

We might imagine that in this simple example, we could employ a scene class that contains an array of CBoxNode structures.

Note: The code we discuss in this first section is purely for teaching purposes and will not be used by any of our applications. It is used to solidify concepts. Later in the lesson we will develop the actual code that our applications will use.

```
class CScene
{
    public:
    CScene ( CPolygon **ppPolygons , long PolygonCount);
    CBoxNode  BoxNode [49] ;
};
```

In this very simple example, the scene has a constructor that accepts an array of CPolygon pointers and the number of pointers in this array. This is where we would pass in the array of 100,000 polygons we have loaded from our town file. This function would be responsible for calculating the bounding box of the entire scene and then generating the 7x7 (49) CBoxNode structures.

The CScene class has an array of CBoxNode structures, one for each bounding box we will create to represent the level. This structure might be defined like so:

```
class CBoxNode
{
   public:
   D3DXVECTOR3       m_vecBoxMin;
   D3DXVECTOR3       m_vecBoxMax;
   CPolygonContainer m_PolyContainer;
};
```

This simple class stores the minimum and maximum extents of the bounding box this node will represent, and a polygon container. We might imagine this to be a simple class that wraps an array of polygon pointers and exposes member functions for adding and retrieving polygon pointers to/from that array. We provided many classes that did similar things in Module I and of course, STL vectors could be used for the same task. This container will initially be empty for each box node. We will discuss why we will need this container stored in the box node class in a moment.

The job of the scene's constructor would be to compile a bounding box for the entire scene. This is done by simply testing the vertices of every polygon passed in and recording the maximum and minimum x, y, and z components found. The first section of such a function is shown below.

```
CScene::CScene ( CPolygon **ppPolygons, long PolygonCount )
{
   // Set master scene bounding box to dummy values
   D3DXVECTOR3 vecBoundsMin(  FLT_MAX,  FLT_MAX,  FLT_MAX );
   D3DXVECTOR3 vecBoundsMax( -FLT_MAX, -FLT_MAX, -FLT_MAX );
   int i , k , j;

   // Loop through each polygon in list and calculate bounding box of entire scene
   for ( i = 0;  i < PolygonCount; i++ )
   {
     // Get pointer
     CPolygon * pPoly =  ppPolygons[ i ];
```

```
    // Calculate total scene bounding box.
    for ( k = 0; k < pPoly->m_nVertexCount; k++ )
    {
      // Store info
      CVertex * pVertex = &pPoly->m_pVertex[k];

      // Test if this vertex pierces the current maximum or minimum
      // extents if adjusting extents if necessary
      if ( pVertex->x < vecBoundsMin.x )  vecBoundsMin.x = pVertex->x;
      if ( pVertex->y < vecBoundsMin.y )  vecBoundsMin.y = pVertex->y;
      if ( pVertex->z < vecBoundsMin.z )  vecBoundsMin.z = pVertex->z;
      if ( pVertex->x > vecBoundsMax.x )  vecBoundsMax.x = pVertex->x;
      if ( pVertex->y > vecBoundsMax.y )  vecBoundsMax.y = pVertex->y;
      if ( pVertex->z > vecBoundsMax.z )  vecBoundsMax.z = pVertex->z;

    } // Next Vertex

  } // Next Polygon
```

At this point we have the bounding box for the entire scene. We will now carve this box into 49 boxes (7 rows of 7 columns) and assign the resulting bounding boxes to the 49 CBoxNode objects of the scene class. This code first divides the width and depth of the scene's bounding box (calculated above) by 7 along the X and Z extents to carve the bounding box into 49 sub-boxes in total.

> **Note:** In this example we are subdividing the scene along the X and Z extents, which can be seen if you look at the previous diagram and imagine that we are looking down on the scene from a bird's eye view.

The two loops shown below are used to loop through each row (i) and each column (k) and calculate the index of the current CBoxNode object that we are processing. We use the formula (i * 4) + k. If *i* is the number of rows of boxes that we want (7) and *k* is the number of columns we want on each row (7) then we can see that when *i*=2 and *k*=1 we are calculating the bounding box for the ((2*7)+1) = 15th CBoxNode object in the scene's array. Remembering that arrays use zero-based indexing, we can see how this CBoxNode object would represent the 2nd box (*k*) in the 3rd row (*i*) of our scene.

```
float StepX  = ( vecBoundsMax.x - vecBoundsMin.x)  / 7.0f ; // Seven Columns
float StepZ  = ( vecBoundsMax.z – vecBoundsMax.z)  / 7.0f ; // Seven Rows

for ( i = 0; i < 7; i++ )
{
  for ( k = 0 ; k < 7 ; k++ )
  {
    // Get the index of the Box Node we are currently calculating
    int nBi = ( i * 7 ) + k ;

    BoxNode[ nBi ].m_vecBoxMin.x  =  ( k * StepX );
    BoxNode[ nBi ].m_vecBoxMax.x  =  BoxNode[nBi].m_vecBoxMin.x + StepX;
    BoxNode[ nBi ].m_vecBoxMin.z  =  ( i * StepZ );
    BoxNode[ nBi ].m_vecBoxMax.z  =  BoxNode[nBi].m_vecBoxMin.z + StepZ;
    BoxNode[ nBi ].m_vecBoxMin.y  =  vecBoundsMin.y;
    BoxNode[ nBi ].m_vecBoxMax.y  =  vecBoundsMax.y;
  }
}
```

At this point in the scene's constructor we have uniformly subdivided the bounding volume of the entire scene into 49 separate bounding volumes that together represent the space that was described by the scene's original bounding box. These bounding volumes are now stored in the scene's CBoxNode array.

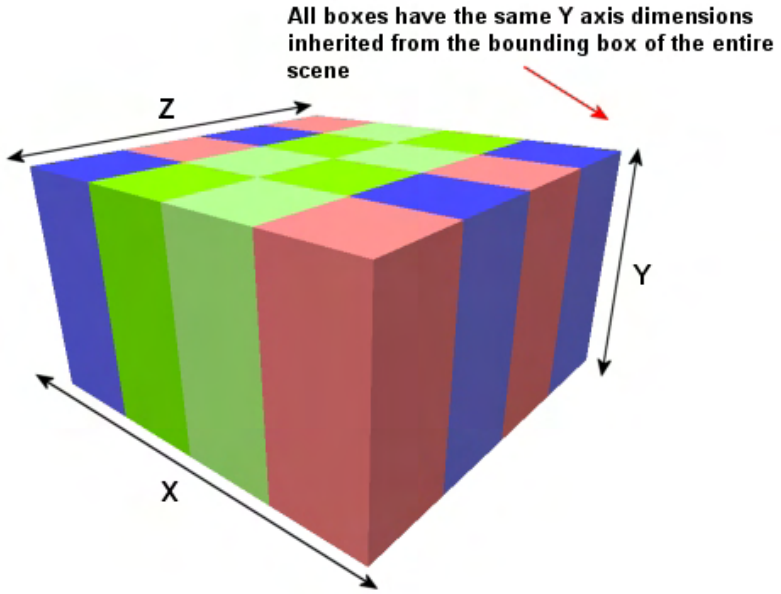


Figure 14.3

Because we are only dividing the scene along the X and Z axes in this example, we assign the minimum and maximum Y axis extents of the bounding box recorded for the entire scene to each of the CBoxNode objects. This means that all 49 sub-boxes generated will all have identical heights along the Y axis. Those Y axis extents are inherited from what we can refer to as the parent bounding box (i.e., the bounding box calculated for the entire scene at the top of the function). Figure 14.3 illustrates the spatial partitioning that has been applied to the scene's original bounding box (albeit for a 4x4 example in this case).

Having 49 empty bounding boxes is fairly useless until the next aspect of our scene preparation is performed. We must now loop through each bounding box and, for each box, loop through each polygon in the scene and test which ones intersect it. For a given bounding box, any polygons that intersect it or are contained completely inside it have their pointers added to the CBoxNode's polygon container. At the end of the following code, our constructor will be complete and when the scene is initialized, we will not only have 49 bounding boxes, in each box node structure we will have a list of polygons that are contained (or partially) contained within that region of space.

```
for ( k = 0 ; k < 49 ; k++ )  // 49 box nodes in this example
{
  // loop through each box
  CBoxNode * pNode = &BoxNode[ k ];

  // Loop through each polygon and test for intersection with box
  for ( j = 0;  j <  PolygonCount; j++ )
  {
    // Get pointer
    CPolygon pPoly =  ppPolygons[ j ];

    // If polygon intersects add its pointer to the box node's container
    if ( AABBIntersectPolygon(   pNode->m_vecBoxMin ,
                                 pNode->m_vecBoxMax
                                 pPoly ))
    {
       pNode->m_PolyContainer.AddPolygonPointer( &pPoly );

   }  // End if Intersect
```

```
    } // End Each Polygon
  } // End Each box
} // End function 'CScene::Constructor'
```

> **Note:** Again you are reminded that this code is not used by any of our applications. It is being used to demonstrate a concept and nothing more. We are using classes with methods here that we have never written, nor do we intend to. Please view these listings as being semi-pseudo code for the time being.

As you can see by examining the final piece of the function code, it involves nothing more than testing each bounding box node to see whether or not any of the scene polygons intersect with it. If so, then a pointer to that polygon is added to the polygon list in the box node. This code assumes that the polygons are stored in each box in a container class that has a member function called AddPolygonPointer, which simply adds the passed pointer to the end of polygon pointer array maintained by the container.

In this particular example, if a polygon is found to be contained in multiple boxes, we store a copy of its pointer in each. This causes no real harm, although when performing subdivision in this way we often wish to make sure that we do not test the same polygon more than once. For example, if we performed a collision query and found that three boxes were intersected by the bounding volume of our swept sphere, we might collect all the polygons from all those boxes and add them to a potential colliders list that is then passed onto the narrow phase. However, it is possible that a large polygon may have intersected all three of those box nodes and as such would be added to the potential colliders list three times. We can certainly work around this in the intersection testing case by testing to see if the given polygon has already been tested for collision so that we do not test it again. However, we must bear in mind that if we have to employ complex loop logic to determine such things, this could possibly outweigh a lot of the savings that the broad phase has introduced.

A common technique when using such a scheme is to allow the polygon structure to store a 'current time' member. When the narrow phase steps through the list, it will process a polygon and set its time member to the current time of the application. If it encounters a polygon in the list with the same time as the current application time, then this must be a pointer to a polygon it has already processed and will skip it. However, this still adds a per-polygon conditional test in the case where the bounding volumes contain polygon data. Under certain circumstances, this can harm performance as well. However, this might be a good strategy for dynamic objects that are assigned to nodes and can have this test performed on the per object level where the conditional tests would introduce negligible overhead. Things are also less simple when spatial partitioning for your rendering pipeline. We desire the vertices of the renderable versions of the geometry to be stored in vertex buffers (static buffers, preferably) rather than CPolygon structures. As such, we have no good way of setting a variable on a per-polygon basis like this and neither would want to. In this case, we may be forced to render the same polygon multiple times.

You will see later that one way to get around this problem is by clipping our scene polygons during the compilation of the spatial tree. In this scenario, a bounding volume will contain only polygons that fit fully inside its volume. If, during the compilation of the bounding volumes, a polygon is found to be inside two boxes at the same time, the polygon will be split into two pieces at the box boundary and the separate fragments will be stored in the respective bounding box. We will also implement an elegant solution for the case when you do not wish clip the polygon data and increase the polygon count of the scene, which will be explained later. For the sake of this current discussion, we will just assume that we

have assigned any polygon spanning multiple nodes to all nodes in which it is contained and will just live with the cost of having it exist multiple times in the potential colliders list.

The kind of spatial subdivision we have shown above would be performed once when the scene is first loaded. Alternatively, the compilation could be done at development time and the scene saved out to file in its subdivided format. Compiling the scene organization at development time is often necessary when we start subdividing scenes with hundreds of thousands or more polygons into thousands of bounding boxes. In these instances, the subdivision of the scene can take quite a long time. Remember, if we are testing hundreds of thousands of polygons against thousands of AABBs, even on today's microprocessors, compilation time will not be instant. If performed at run time when the level is first loaded into the computer's memory, this might subject the user of our application to an unacceptable delay. Therefore, it is often common for a file containing scene data to also contain the information in a subdivided format. Using our simple 49 box subdivided scene, we might store the data in the file such that the file contains 49 bounding box data structures and a list of polygons with accompanying bounding box index values describing which bounding boxes each polygon belongs in. The loading code could simply pull the bounding box data out of the file and use it to create the box nodes. The polygon data could then be extracted and its box index list for each polygon examined and used to assign the polygon to its pre-determined bounding box(es).

14.1.1 Efficient Polygon Queries

Let us now return to the problem of trying to determine which polygons in our level should be considered for the narrow phase during a collision test. Recall that we had assumed the compilation of an AABB around our swept sphere and now wish to calculate the polygons that intersect that bounding box and thus should be added to a potential colliders list. Figure 14.4 reminds us of the current location of our bounding volume within the scene.

We have assumed a level size of 100,000 polygons which each originally had to be tested against the AABB. Now our scene had been divided into 49 bounding boxes and as such, the broad phase requires nothing more than performing an AABB/AABB intersection test between our swept sphere's bounding volume and the 49 bounding volumes of the level. AABB\AABB tests are very cheap to perform and we can narrow down the number of polygons that need to be passed to the narrow phase to just the relative handful of polygons that have been assigned to the bounding boxes that our swept sphere's bounding box intersects.

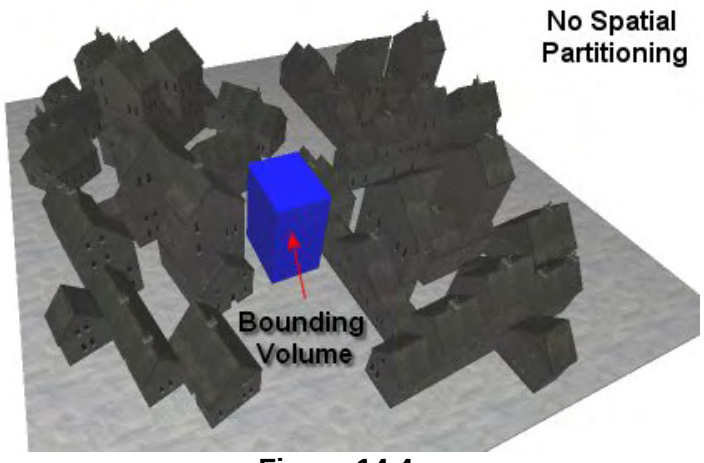


Figure 14.4

As Figure 14.5 demonstrates, after performing only 49 AABB\AABB tests, we find that only two of the scene's bounding volumes intersect our swept sphere's AABB. We can then quickly fetch the polygons assigned to each of these two box nodes and add them to the potential colliders list.

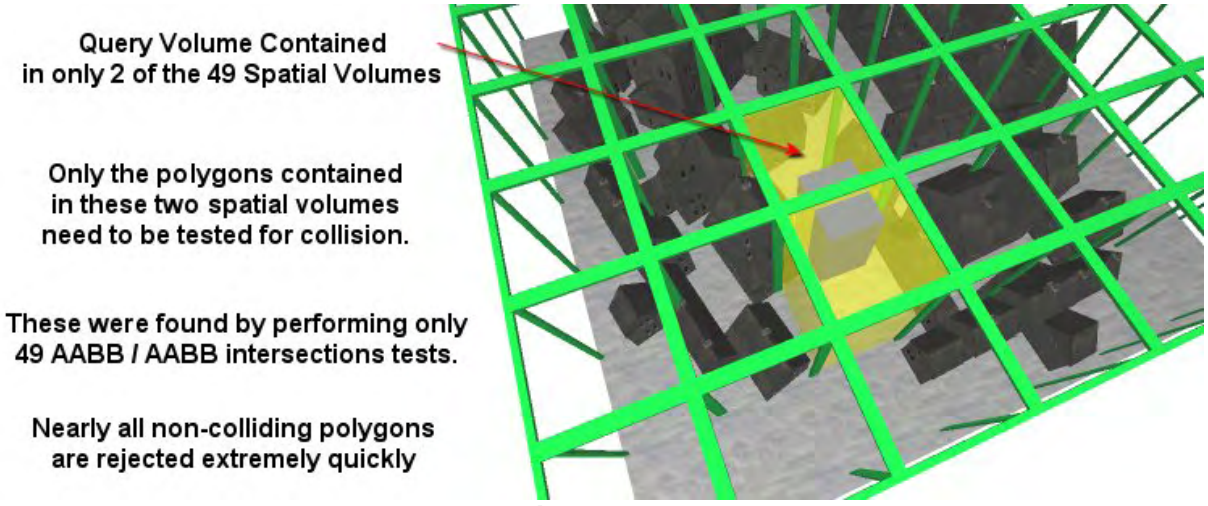


Figure 14.5

If we assumed for the sake of demonstration that the polygons were evenly distributed throughout the level, each box would contain 100,000 / 49 = 2040 polygons.

Figure 14.6

Thus with only 49 very efficient AABB/AABB tests, we have just rejected 47 of these boxes and the 2040*46 = 95,880 polygons contained within. Not bad! Only 2*2040 = 4080 polygons would need to be passed to the narrow phase for closer inspection by the collision routines.

As far as the narrow phase is concerned, the world is only as large as those two boxes since these are the only polygons it will need to test. Through the 'eyes' of our narrow phase, the world would look what is shown in Figure 14.6.

Below we see the code to a function that would implement a broad phase in this manner called GetPotentialColliders.

This function could be called by the collision system as its broad phase when the sphere has moved. It would be passed a pointer to the scene and the minimum and maximum extents of the AABB that is bounding the swept sphere. As its final parameter it takes a pointer to an object of type CPolygonContainer. We will assume for now that this is a simple class that encapsulates a polygon array and allows polygon pointers to be added to its array via its member functions. The container is assumed to be empty when this function is called and will be used to store the potential colliders it collects. When the function returns, the collision system will be able to pass this container to the narrow phase.

The function has been deliberately hardcoded for our 49 box example for the purpose of demonstration (again, we will not be using any of this code in practice). It also uses a function called AABBIntersectAABB which is an intersection testing function which we will discuss shortly. The AABBIntersectAABB function takes the extents of the two bounding boxes that are to be tested as its parameters and returns true if the two are intersecting.

```
void GetPotentialColliders( CScene *pScene,
                            D3DXVECTOR3 BoxMin,
                            D3DXVECTOR3 BoxMax,
                            CPolygonContainer *pContainer)
{
     for ( ulong i = 0; i < 49 i++ )
     {
          CBoxNode *pNode = &pScene->BoxNode[i];
          if ( AABBIntersectAABB( BoxMin, BoxMax,
                                  pNode->m_vecBoxMin,
                                  pNode->m_vecBoxMax )
          {
              for ( ulong k = 0; k < pNode->m_PolyContainer.GetSize(); k++ )
              {
                  pPoly = pNode->m_PolyContainer.GetPolygonPointer(k);
                  pContainer->AddPolygonPointer( pPoly );
              }
          }
     }
}
```

As you can see, this function simply tests the bounding volume passed in against the bounding volumes for the scene. If any scene bounding volume is intersecting, the polygons are fetched from the box node's container and added to the potential collider container. When the function returns, the passed container will contain all the polygons that should be more closely examined in the narrow phase.

The application of spatial partitioning is obvious when discussed in the context of running polygon queries such as those often required to be performed by a collision detection system. But does spatially partitioning the scene provide us with any other benefits? Does it possess the ability to speed up rendering as well? It certainly does.

14.1.2 Efficient Frustum Culling

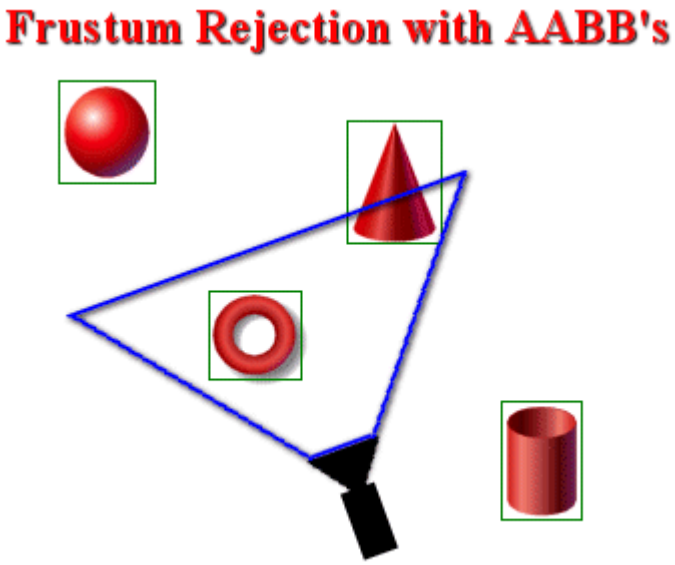


Figure 14.7

In Module I we learned how to perform AABB/frustum culling and used it to reject non-visible meshes prior to being passed through the rendering pipeline. Rather than passing a mesh containing thousands of polygons to the DirectX pipeline and allowing DirectX to determine which polygons are visible and which are not, we decided to help the process along by performing mass polygon culling using bounding boxes.

While the Direct X pipeline is good at what it does and has some very efficient code to reject polygons from the pipeline as soon as they are found to lay outside the view frustum, this culling still has to be performed on a per-polygon basis. If your scene consists of 100,000 polygons, the pipeline will need to perform 100,000 polygon/frustum tests each time the scene is rendered. Even worse is the fact that these polygons all need to be transformed into homogeneous clip space before the culling can commence. While hardware is very fast, this can become a bottleneck with scenes comprised of a large number of polygons.

In Module I we also learned that we could greatly increase our rendering performance if we assign each mesh an AABB and test it against the world space frustum planes prior to rendering that mesh. If we can detect that a mesh's AABB is completely outside the view frustum, we do not have to bother rendering it. Figure 14.7 shows four meshes and a camera along with its frustum. In this example we can see that only two of the four meshes need to be rendered, as the cylinder and the sphere are positioned well outside the camera's frustum.

Let us now return to our little town scene consisting of 100,000 polygons. We are assuming for the time being that this data is loaded as one big polygon soup. It should now be obvious that since we have subdivided our scene into AABBs, the same frustum culling strategy can be employed for our scene's bounding boxes, allowing us to frustum cull geometry in (theoretically) even bigger chunks than the mesh based approach covered in Chapter 4. Any one of our scene's bounding boxes may contain many polygons or perhaps many separate meshes, all of which could be frustum culled from the rendering pipeline with a single AABB / Frustum test.

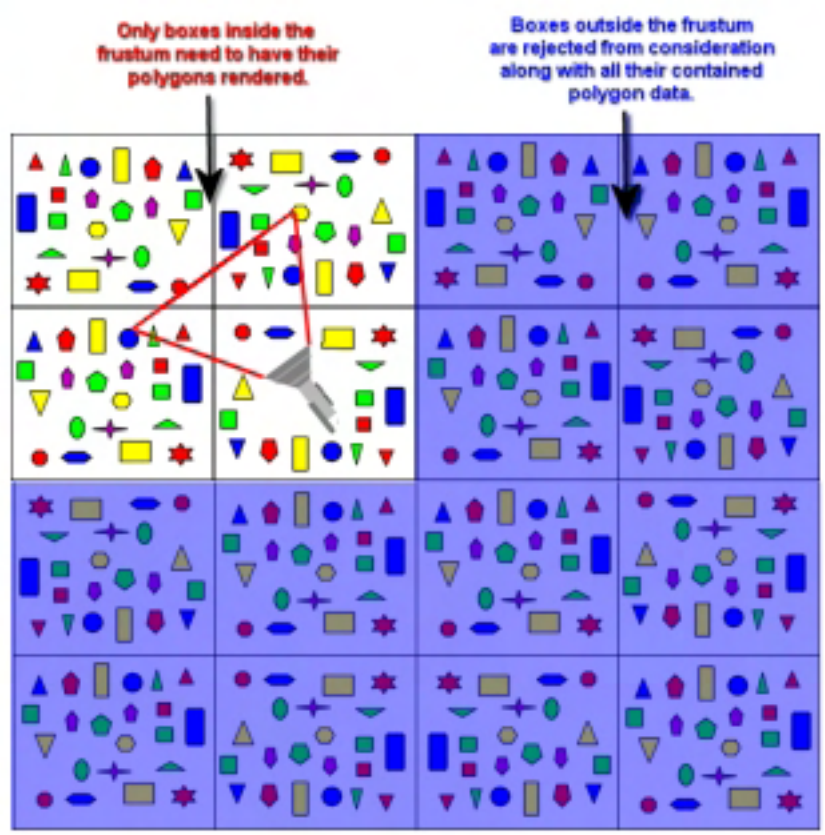


Figure 14.8

Using a 16 box scene example, this time we can see that in Figure 14.8 the camera is positioned and oriented such that only the polygons in 4 of the 16 boxes could possibly be visible. The frustum is shown as the red lines extending out from the camera and we can see that only the four top left boxes intersect the frustum in some way.

Using exactly the same frustum/ AABB code we covered in Chapter 4 of Module I, we have the ability to mass reject quite a bit of scene geometry. If the depicted scene consisted of 100,000 polygons equidistantly spaced such that each of the 16 boxes contained 6250 polygons, we can see that before we render the scene, we would test the 16 AABBs (box nodes) of our scene against the camera's frustum and find 12 of the boxes to be completely outside of it. With just 12 efficient tests, we have rejected 12 * 6250 = 75,000 polygons. Since only four of the boxes intersect the frustum we would only need to transform and render 4 * 6250 = 25,000 polygons (1/4th of the scene).

The following function shows a strategy that might be used to render the 16 box scene shown above. Assuming the scene has already been compiled into its 16 box format and that we are using an AABB / Frustum function which is a member of the camera class called IsAABBInFrustum (the code of which was covered in Module I), the rendering strategy might look something like this:

```
void CScene::Render( CCamera *pCamera )
{
      // Loop through each box
      for ( int i = 0 ; i < 16 ; i++ )
      {
         // Is this box inside the frustum or partially inside the frustum
         if ( pCamera->IsAABBInFrustum( &BoxNode[i].m_vecBoxMin ,
                                        &BoxNode[I].m_vecBoxMin ) )
         {
            // It's a visible box so render all the polygons stored here
            for ( k = 0 ; k < BoxNode[i].m_PolyContainer.GetPolygonCount(); k++ )
            {
                  // Get a point to the current polygon we wish to test
                  CPolygon * pPoly = BoxNode[i].m_PolyContainer.GetPolygon( k );

                  // Render Polygon
```

```
                    RenderPolygon ( pPoly );
                }
            }
        }
}
```

Of course, the above code is for example only and you certainly would not want to render all the polygons assigned to a given box one at a time as that would negate any performance gained from batch rendering. Later we will implement a system that will provide our code the benefits of spatial rejection during rendering but do so in a way that makes it hardware efficient (in fact, this will be the main focus of the next lesson). Hopefully though, even this simple code demonstrates the benefits of spatial partitioning, not only for performing per-polygon queries on the scene's data set such as intersection testing, but also for efficiently rendering that dataset when much of the scene lay outside the frustum.

Indeed it may have occurred to you as you were considering this concept that frustum culling is simply another form of intersection testing. That is, we are colliding one volume (our frustum) with some other volume (the boxes) to collect a set of potentially *visible* polygons. Those polygons are then passed on to a narrow phase (the DirectX pipeline) where the actual visible polygons will be rendered and the others fully or partially clipped away.

Finally, it should be noted that this same system can be used whether your scene is represented as a single mesh or comprised of multiple meshes or both. For example, a box node could contain a list of mesh pointers instead of a list of polygons. Assigning a mesh to bounding boxes could be done by simply figuring out which scene bounding boxes it intersects during the compile process. Later on, we will provide support for both of these systems. That is, the static geometry we load from an IWF file will be spatially partitioned and stored in bounding volumes at the per-polygon level, while dynamic objects such as actors will be assigned to bounding volumes at the object level. In our case, the nodes of our spatial tree will have an understanding of what polygons are contained inside them and our dynamic/detail objects will have knowledge of which spatial nodes they are contained within (more on this later).

Although we have seen how vital spatial partitioning is and how it can improve polygons querying efficiency, the system described so far is really quite inadequate. We have been using very simple numbers to demonstrate the point, but in reality, game scenes tend to be quite large and we ultimately wish to send as few polygons to the narrow/rendering phases as possible. This means that we generally wish to have far fewer polygons stored in each box node. Logically this also means subdividing the scene into many more box nodes. It is more probable that we would want to divide the average scene into thousands of boxes so that each box only contains a handful of polygons. This would mean for example, that if our swept sphere's AABB was found to be intersecting two box nodes, we would only be sending a handful of polygons to the narrow phase, instead of 4000 as in our previous example.

> **Note:** To be fair, given the performance of today's rendering hardware, larger node sizes are not necessarily going to be a bad thing. However, for CPU oriented tasks like collision detection, smaller batch sizes are definitely preferable.

Now that we basically understand what spatial portioning is and how bounding volumes can have scene polygon/mesh data assigned to them, we can take this conversation to the next level and discuss hierarchical spatial partitioning.

14.2 Hierarchical Spatial Partitioning

While the strategy discussed in the last section was great for demonstrating the basic process, it left much to be desired. In our previous example we simply used the value 49 as the number of boxes to divide our scene into, but in reality you would probably want to divide your scene into many more regions than this. If a level contained 100,000 polygons for example, each box would contain 2040 polygons. Even if the query volume only intersected a single box, that is still far too many polygons to send to the narrow phase of our collision system. Remember, given all that must be accomplished in any given game frame, our polygon queries will need to be executed within mere microseconds if frame rates are to remain acceptable.

It would seem that in order to make this strategy more efficient, we would want to divide a typical scene into many more boxes than this. For example, we might decide that we do not want any more than 100 polygons to be contained within a single box node. This would require the scene to be divided into 1000 boxes in our previous example. While this obviously means the initial scene would take much longer to compile (as we would now have to test each polygon against 1000 AABBs instead of just 49) this is not a problem since this can be done as a pre-process at development time and the information saved out to file as discussed earlier. The real problem now is that the broad phase has now been made much more expensive because it would have to test 1000 AABBs instead of just 49 to collect potential colliders for the narrow phase. This is obviously going to be quite a good bit slower. Also, it means that the rejection of a single box now only rejects 100 polygons instead of 2040 polygons, so each AABB/AABB test has less bulk rejection impact. On the bright side, after we have performed all the AABB/AABB tests in the broad phase, we would be left with a considerably smaller polygon list that needs to be passed to the narrow phase.

So while the dividing of the scene into many more smaller boxes has reduced the number of potential colliders that are collected and passed to the narrow phase of our collision system (or rendered in the case of frustum culling), much of CPU savings is lost to the larger number of AABB/AABB tests we have to perform. To be sure, this would still be much faster than just passing every polygon to the narrow phase, but unfortunately it is still nowhere near fast enough.

So how can we divide the scene into many more boxes with small clusters of polygons stored in each box while still maintaining the ability to reject the individual boxes from queries extremely quickly? We use a spatial hierarchy.

We discussed hierarchies in Chapter 9 when we learned how to load in hierarchical X files. The hierarchies in that chapter established spatial relationships between meshes in an X file. The hierarchy could be thought of as a tree structure where each node of the tree was a D3DXFRAME which had pointers (or branches) to other sibling and child frames in the tree. Some of the frames actually contained meshes (via mesh containers) and using the tree analogy, we can think of these nodes as being the leaves of the tree. When we examined the D3DXFrame hierarchy, the parent-child relationship between the frames in the tree established how meshes in the scene were related. If rotation was applied to a node in the tree, the children of that node also inherited the rotational change. This allowed us to store a car model as separate meshes, where the wheel meshes could be rotated independently from the car body stored in a parent node (frame). Any movement or rotation applied to the car body node was

also inherited by the wheel nodes so that when the car body was moved, the wheels always moved with the car body and maintained their correct spatial relationship.

Hierarchies are used so often in game development and in so many different areas that it is hard to imagine creating a 3D application without them. Spatial partitioning data can also be represented as a hierarchy which makes finding the regions of the scene that need to be queried or rendered extremely efficient. If we think back to our D3DXFrame hierarchy from Chapter 9, we can easily imagine that we could assign each frame in the hierarchy a bounding box which was large enough to contain all the meshes below it in the hierarchy (i.e., meshes belonging to direct and indirect child frames). If we were to do this, then the frame hierarchy could be traversed from the root node down during a frustum culling pass and as soon as a frame was found that had a bounding box that was situated outside of the frustum, we could stop traversing down that section of the hierarchy and reject the frame along with all of its descendants from rendering consideration. This rejected frame may have had many child frames which each had many child frames of their own with meshes attached. By simply rejecting the parent frame of the hierarchy, we reject all of the children and grandchildren, etc. with one simple test at the parent node. Of course, this strategy assumes that the parent node's bounding volume has been calculated such that it encompasses the bounding volumes of all its child nodes, and so on right down the hierarchy (i.e., bounding volumes are propagated *up* through the tree, getting larger as more children join in).

It might have occurred to you that this same hierarchical technique could be employed with spatial partitioning. Rather than simply divide the scene into 49 bounding boxes and store all 49 boxes in the scene class, we could instead recursively divide the level, creating a tree or hierarchy of bounding volumes. In this example, the bounding volume being used is an AABB, but we could use other bounding volumes also (spheres are another popular choice).

The following images demonstrate the division of our same example scene hierarchically. The end result in this example is actually a scene that will be divided into 64 bounding boxes (at the leaves of the tree). Because these bounding volumes are connected in a tree like structure, queries can be done extremely quickly.

Forgetting about how we might represent this subdivision technique in code for the moment, let us just analyze the properties of the images. The scene is as before, a large rectangular region of polygon data. The bounding box of this entire region would be considered the bounding box of the entire scene.

When the scene constructor is called we might imagine calculating this bounding box node and storing it in the scene class. This will be the root node of the tree and the only node pointer the scene will store. This concept is not new to us. Recall that our CActor class stores only the root frame of the hierarchy. The rest of the frames in the hierarchy can be reached by traversing the tree from the root. We are now imagining a similar relationship between our spatial nodes. To accommodate this arrangement we can upgrade BoxNode to store pointers to child BoxNode structures.

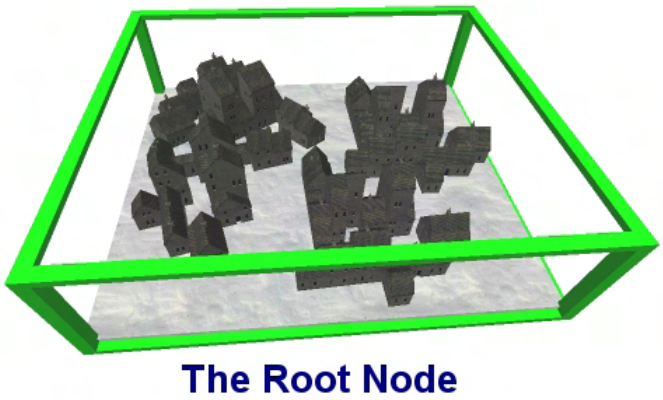


Figure 14.9

```
class CBoxNode
{
   public:
   D3DXVECTOR3       m_vecBoxMin;
   D3DXVECTOR3       m_vecBoxMax;
   CPolygonContainer m_PolyContainer;
   CBoxNode *        m_Children[4];
};
```

Notice that we have now added an array of four CBoxNode pointers to the CBoxNode class. This means each node will have four child nodes.

> **Note:** A spatial tree that divides the parent node space into four at each node is called a *quad-tree*. Thus, what we are looking at above is a Quad-tree node. You will discover later how the only real difference between an oct-tree, quad-tree, and kD-tree is simply how many children each node in the tree has. For the sake of explaining hierarchical spatial partitioning in this section, we will use a Quad-tree in our examples.

As Figure 14.9 shows, the first step in compiling our spatial tree is to compute the bounding box of the entire scene. The bounds calculated will be stored in a CBoxNode structure whose pointer is stored in the scene class. This will be the root node of our tree.

In our example we will be using a quad-tree to partition space which essentially means each node in our tree has four direct child nodes and therefore, at any given node in the tree, its immediate children partition its bounding volume into four equal sub-volumes (called *quadrants*).

With the root node created, we enter a recursive process to build the rest of the tree. At each recursive step (starting at the root) we are passed a list of polygons contained in that node's volume and have to decide whether or not we wish to partition this space any further. If we do decide that the bounding volume of the current node is sufficiently small or that the number of polygons that have made it into this volume are so low in number that further spatial subdivision of this region would be unnecessary, we can simply add the polygons in the list to the node's polygon container and return. We do not bother creating any children for this node. It is essentially a node at the end of a branch and is therefore referred to as a *leaf node*. Leaf nodes are the nodes at the ends of branches which contain polygon/mesh data. The branch nodes are sometimes called *normal nodes* or *internal nodes* or *inner nodes* or simply just nodes (versus leaves).

However, if we do wish to further partition the space represented by the node's bounding box, we divide the node's bounding box into four sub-boxes. We then create four child nodes and assign them the bounding boxes we have just calculated. We then classify all the polygons in the list of polygons that made it to this node against the four bounding boxes of the child nodes. This allows us to create four sub-polygon lists (one per child). We then recur into each of the four child nodes, sending it its polygon list so that this same process can occur all the way down the tree. Notice that when a node has children, it stores no polygon data (at least in this vanilla implementation) and is considered to be a normal node (as opposed to a leaf node at the end of a branch of the tree).

In Figure 14.9 we generated the root node and its bounding volume. Let us step through the process of building the quad-tree with images to solidify our understanding. Please note that in these examples we are still only carving up the scene into 49 boxes at lowest level of the tree since this makes the images much easier to decipher. You can imagine however that this same process can be performed to divide the scene into 1000s of boxes which are hierarchically arranged.

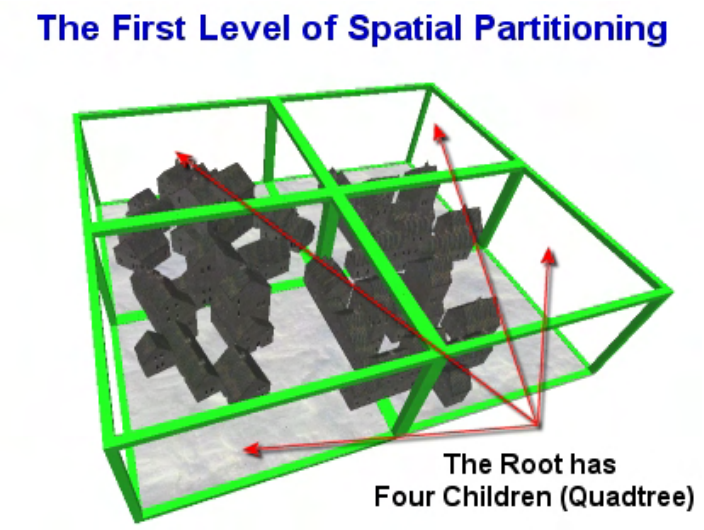


Figure 14.10

We would first process the root node, being passed the list of polygons for the entire scene. We would determine that the node's bounding box is very large and therefore its space should be further partitioned. The polygon list would contain a large number of polygons and we will not be creating a leaf at this node. Remember, we generally want to store only a handful of polygons in the leaf nodes so that we minimize the number of polygons that are collected and sent to the narrow phase when a leaf node's volume is intersected.

Thus, we create four child nodes and attach them to the root node. We would then divide the bounding box of the root node uniformly into four sub-boxes, each of which describes a quadrant of the parent volume (Figure 14.10). Each child node would be assigned one of these child bounding volumes. The list of polygons would then be tested against the four bounding volumes and a list compiled of the polygons contained in each box (four lists). It is at this stage that any polygons found to be spanning a box boundary could be clipped so that the polygon list compiled for each child will contain only the exact polygons that are contained inside its volume. If clipping is being used, a child node will never be sent a polygon list that contains polygons that span the boundary of its bounding box. We would then recur into each child passing in the polygon list that was compiled for it by the parent.

Remembering that this is a recursive process, Figure 14.11 shows what our tree would look like after we have stepped into each child of the root and subdivide their space into four, with polygon lists created for each child node. In reality, the repeated subdivision of each child branch will typically be performed at one time, but we took a bit of artistic license here to demonstrate the subdivision.

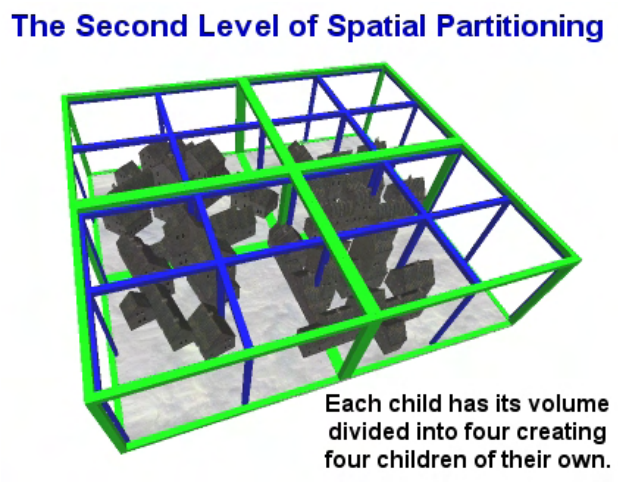


Figure 14.11

If the children of the root node's children are where our recursive process was to end, Figure 14.11 would show the total level of subdivision. As we can see, the root node's bounding box have been divided into four quadrants representing the volume of its four immediate children. Then, each child node of the root, has four children of its own which further partition its volume into four more sub volumes. The blue bounding boxes in Figure 14.11 show the bounding volumes for nodes positioned at the 3rd level in the hierarchy. If we decided to stop partitioning here, then each blue box would be a leaf node and would store a list of polygons that are contained within its volume; the nodes at the first and second level would not. They would just store a bounding volume and pointer to the child nodes. However, in this example we will not stop partitioning at the 3rd level, but will stop at the 4th level instead.

This means that each blue box node in Figure 14.11 would also have four children of its own as shown by the red boxes in Figure 14.12. In our example we are going to stop subdividing here at the 4th level in the hierarchy (3rd level of partitioning). This means, the 4th level of our hierarchy will contain the leaf nodes where the polygon data will be stored.

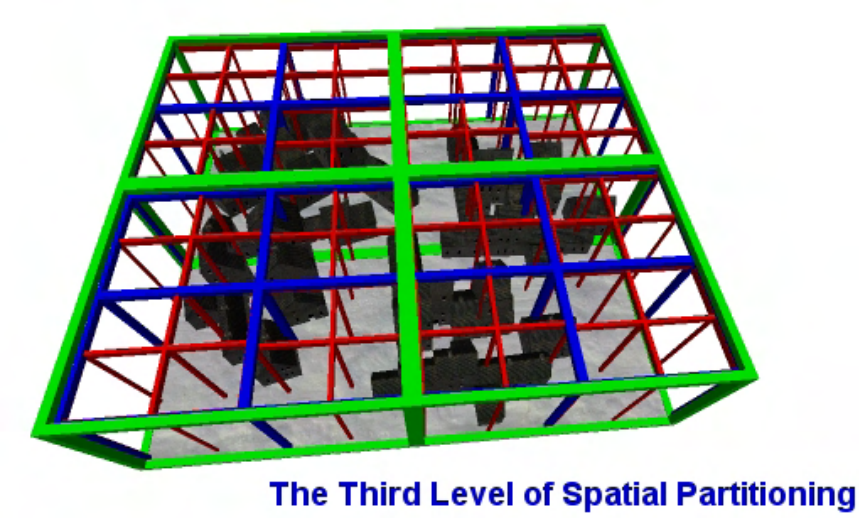


Figure 14.12

Of course, in reality we would subdivide this level to a much greater degree than just 64 bounding boxes, but this subdivision is easy to illustrate in diagrams and subsequently discuss. An actual level that contained 100,000+ polygons would need to be partitioned into hundreds of bounding boxes in order to get optimal collision query performance.

It is also worth noting that in our example we are uniformly partitioning space to get certain sized bounding volumes at the leaf nodes. When this is the case, the quad-tree built will be balanced and every child node at a certain level of the tree will have the same number of children. This is because the scene would be uniformly divided into equal sized leaf nodes. However, often we will not want to subdivide space further if, at any level in the hierarchy, a node contains no polygon data. When this is the case we will generally just make the node an empty leaf (i.e., a node with no children or polygon data). An example of this is shown in Figure 14.13

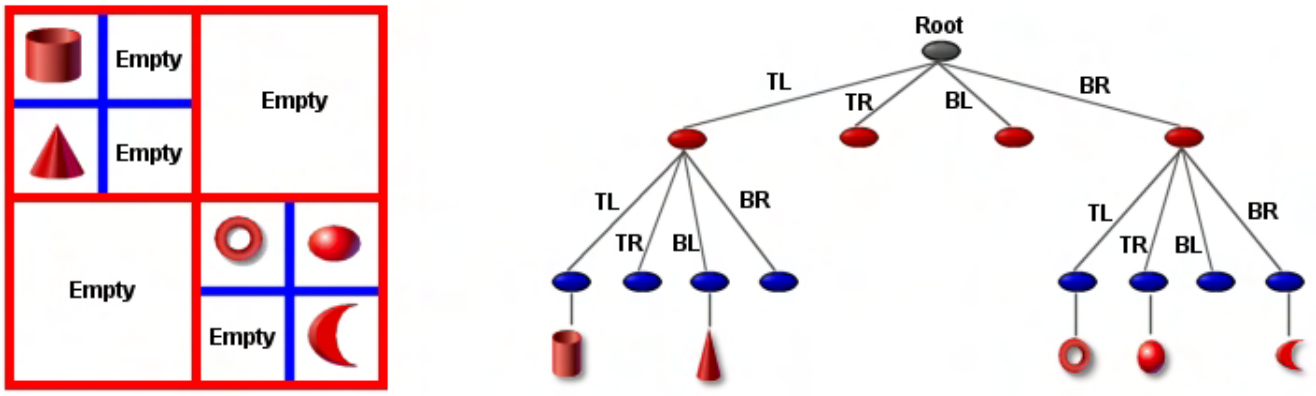


Figure 14.13

On the left side of the image we see the top down view of some shapes being compiled into a quad-tree. At the root node we store the entire bounding box which is then divided into four child nodes representing the four quadrants of the original bounding box. We can see that when building the second level of the hierarchy, the top left and bottom right child of the root contain polygon data and therefore, they are further subdivided (we step into the child nodes and continue our recursion). However, the top right and bottom left children of the root node have no polygon data contained in them so we will not do anything here and return. The fact that we have not created any children for these nodes makes them leaf nodes. Since they have no polygon data, they are empty leaf nodes. On the right of the image we show the shape of the quad-tree. The root has four children and two of those have four children of their own. At the third level, subdivision stops and some of the nodes at this level are empty leaf nodes and some are actual leaf nodes containing polygon data. Notice that we also have empty leaf nodes in the second level too (TR and BL). It would be wasteful in most circumstances to subdivide empty space as these empty volumes will have to be traversed and tested when collision queries are made even though we know that no polygon data will ever be found. The spatial tree shown in Figure 14.13 is not perfectly balanced as the leaf nodes are not all contained at the same level in the hierarchy. Does this matter?

Ideally it would be nice if our tree could be completely balanced so the situation does not arise where some queries may take substantially longer to perform than others merely because the queries are being performed in a section of the tree where the leaf nodes go very deep and more nodes must be traversed before the queried data is located. In a perfectly balanced tree where all leaf nodes existed in the same level of the tree, scene queries would take almost identical times to execute, giving us a consistent query time. We would also use a consistent amount of stack space during the traversal. However, while a balanced tree is a nice thing to achieve, that does not mean we should needlessly subdivide sparse regions of the scene simply to push the sparsely populated or empty leaf nodes down to a uniform level (i.e., alongside leaves from densely populated regions of the scene). After all, making the tree deeper than it need be in many places would essentially be forcing all tree queries to operate under the worst case performance scenario. Tree balance is a more important factor to consider when dealing with kD-trees or BSP trees, as we will discuss in time, but is still something you should bear in mind with all tree implementations. If you find that your scene is taking much longer to query in some area versus others, it may be a balance issue, and in some cases the tree can be made shallower in the troubled areas by not subdividing to such a large degree. It is generally preferable to have you frame rate run consistently at

50 frames per second rather than being at 100 frames per second in some regions and 10 frames per second in others. So consistent query times are certainly desirable where possible.

Using the example quad-tree generated in Figure 14.12, let us see how we might efficiently query the tree for collision information. Once again, at the moment our focus is very much on the implementation of a broad phase for our collision system. We will discuss using spatial trees to speed up rendering later in this lesson and flesh this theory out in the next lesson with a hardware friendly rendering solution.

14.2.1 Hierarchical Queries using Spatial Trees

For this demonstration we will once again assume that we are testing an AABB (surrounding the swept sphere) against the tree starting at the root node. We are interested in finding any polygons in the level that have the potential to collide with the swept sphere. Ideally, we wish to reject all non-potential colliders from consideration as quickly as possible.

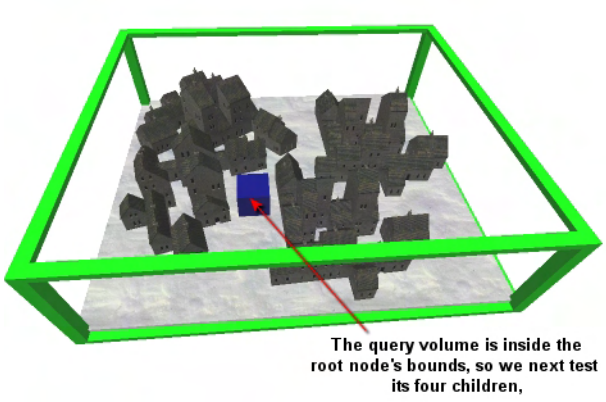


Figure 14.14

The query function would be a simple recursive function which steps through the tree performing intersection tests between the query volume and the bounding boxes of the child nodes of the current node being tested. When an intersection occurs with a child node, the function recursively calls itself to step into that child and continues down the tree. So whenever we find a child node that does not intersect the query volume, we can immediately reject it and all its child nodes (and all the data they contain) instantly.

If at any point we visit a node that has polygon data, we add the data contained in that leaf node to the container being used to collect the polygons for the narrow phase. In Figure 14.14 we show our query volume positioned somewhere within the root node's bounding volume. The first thing we would do is test this bounding volume against the bounding volumes of the root node's four children.

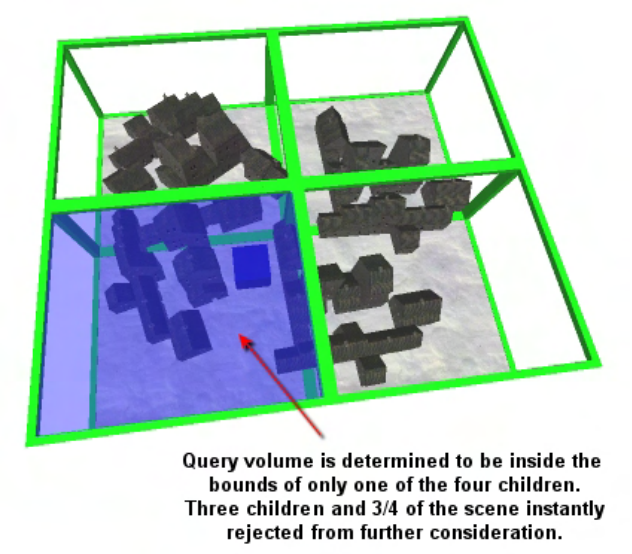


Figure 14.15

As Figure 14.15 illustrates, the query volume would be found to be inside only one of the bounding volumes of the root's children. In this example we can see that it is contained in only the bottom left child node. At this point, we can ignore the other child nodes completely and they do not have to be further traversed. So with four AABB/AABB tests we have just rejected ¾ of the entire scene (assuming even polygon distribution). We know that nothing in the other three child nodes can possibly intersect our query volume, so we ignore those child nodes and their children, and their children, and so on, right down that branch of the tree. If we assume that our level contains 100,000 evenly distributed polygons, we have just rejected 75,000 of them with four simple AABB/AABB tests. Bear in mind that we are only performing very light spatial subdivision here (four levels), but in reality, that step would have probably rejected 100's if not 1000's of leaf nodes and all the polygon data contained within them.

In this instance of the recursive query routine we discovered that the bottom left child of the root was intersected, so the function will now traverse into that node. In the next step we do the same thing all over again. That is, we test the query volume against the four child nodes of the bottom left child of the root. The children of the bottom left child of the root are shown as the blue boxes in Figure 14.16. As you can see, we discover an intersection with only one of the children and the other three child nodes can be ignored. In the first step we whittled away ¾ of the scene, in this second step, we have whittled it down to just a ¼ of that again. With just eight AABB tests, we have rejected 15/16^{ths} of the entire scene. Not bad at all.

Figure 14.16

At this point we are located at the bottom left child of the root and have determined that only its top right child is intersected by our query volume. Therefore, the function would call itself recursively again stepping into this child (the highlighted blue box in Figure 14.16)

Next we would step into the top right blue child (Figure 14.16) and would see that it too has four children of its own. Once again, we would test the query volume against its four children (the four red boxes highlighted blue in Figure 14.17) and determine that the query volume is contained in only two of them. When we traverse down into both of these child nodes we find that they have no children and we have reached the leaf nodes of the tree. We then collect the polygon/mesh data stored in those two leaves and add them to the potential colliders container. We can now return from the function, unwinding the call stack back up to the initial function call made to the root by the application. We would have a container filled with only the potential colliders which can then be passed into the narrow phase. We achieved our goal with only 12 AABB tests. Imagine the savings when the geometry has been divided many levels deep and has created 100's of leaf nodes instead of just 64. For example, if the scene was compiled had 1000 leaf nodes, the first four AABB tests at the root would have instantly rejected 750 of those leaves. This is exactly why hierarchical spatial subdivision is so powerful.

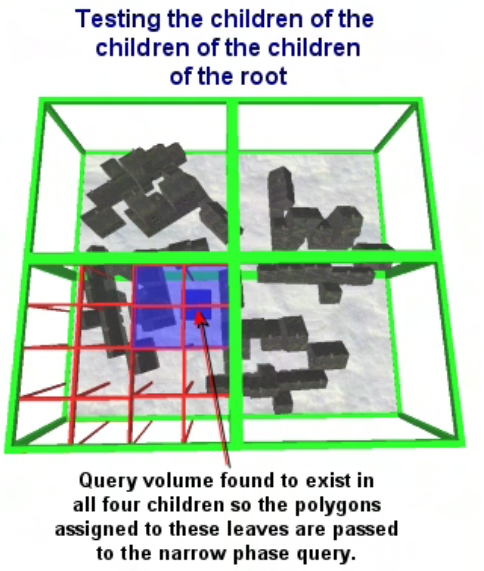


Figure 14.17

14.2.2 Hierarchical Frustum Culling

Let us quickly discuss scene rendering as another example of the benefits of hierarchies. This will only be lightly discussed in this lesson since implementing a hardware friendly rendering system for our spatial trees will be the core subject of the next lesson.

Figure 14.18 shows the first phase of the rendering of this spatial hierarchy. Starting at the root node and traversing down the tree, we can see that at the root node we test its four child nodes (the four quadrants of the entire scene) against the view frustum using AABB / Frustum tests. After these four simple tests we can see that only one of the child nodes of the root intersects the frustum, so the other three child nodes are ignored along with all their children. We have just rejected ¾ of our scene's polygon data from being rendered with these four tests.

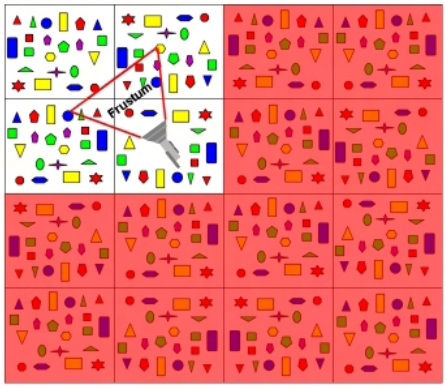


Figure 14.18

The top left child of the root does contain the frustum however, so we will need step down into this node and narrow the set further by testing its four child nodes against the view frustum.

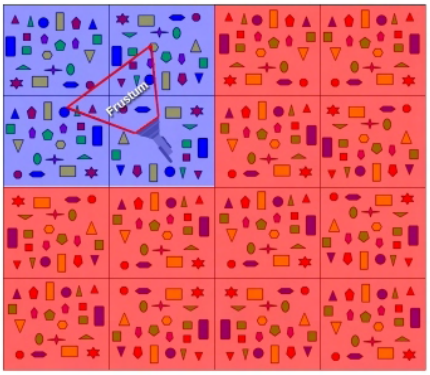


Figure 14.19

When we visit the top left child of the root we must then test to see whether any of that node's child nodes are within the frustum. Once again, we test the frustum against the four bounding boxes of the child nodes, which are shown in Figure 14.19 as the four blue boxes.

One thing that should be obvious looking at Figure 14.19 is that at this level in the tree, all four of the blue child nodes are partially inside the frustum, so we are unable to reject any child nodes at this point. All four child nodes of the top left child of the root will need to be visited to further refine the polygon set that needs to be rendered.

For each of the four blue nodes, we step down and visit their four child nodes. One at a time we determine which of its child nodes have bounding boxes that intersect the frustum. In our simple example scene (Figure 14.20), the children of the blue nodes are also the leaf nodes at the bottom of the tree. These leaf nodes contain the polygon data, so once we find that a leaf node is inside the frustum, we can render the polygons contained there. Alternatively, if you are doing a deferred rendering pass (such as storing the polygons in a queue for sorting purposes) then these are the polygons that should be collected and added to your rendering list.

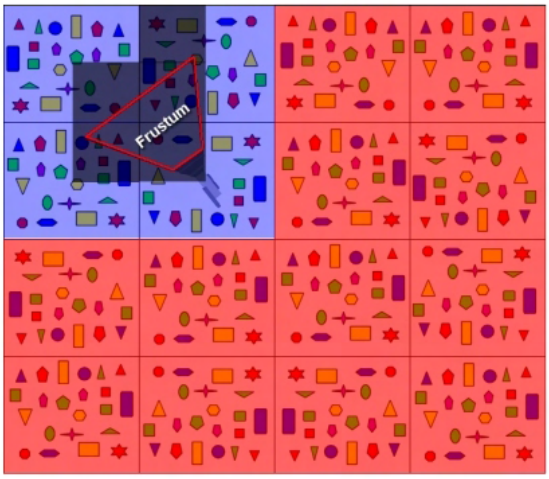


Figure 14.20

As we visit each blue node we determine which of its children are inside the frustum and need to be rendered. We can see in Figure 14.20 that if we start by visiting the top left blue node, only its bottom right child intersects the frustum and it is the only one of its four leaf nodes that needs to be rendered. Then we visit the top right blue node and determine that its top left and bottom left child nodes partially intersect the frustum and need to be rendered. When we visit the bottom left blue node, we see that only one of its four child leaf nodes intersect the frustum (its top right child node). Finally, when we visit the final blue child node on the bottom right, we see that the only child leaf node that is visible is its top left leaf node.

Thus, with the camera positioned as shown in this diagram, only five leaf nodes from a scene comprised of sixty four leaf nodes need to be rendered. By stepping through the hierarchy and performing 24 AABB / Frustum intersection tests, we have rejected 92% of our scene polygons from having to be rendered.

Unfortunately, coming up with a hardware friendly rendering solution is not as trivial as it may appear to be. Even with such large scale rejection of polygon data at our disposal, the strategy of collecting the visible polygons and rendering them must be well thought out so that CPU burden is kept to a minimum. On modern graphics cards, the GPU is very, very fast and you may easily find cases where brute force rendering beats a naïve implementation of hierarchical spatial partitioning. A system that burdens the

CPU will suffer greatly when compared to its brute force counterpart when the entire scene is contained inside the frustum and nothing can be culled. If the entire scene is contained inside the frustum, then everything will need to be rendered anyway. In that case, brute force will always be faster due to the fact that it does not need to perform any tree traversals. So what are our options?

One implementation you might come across in your research steps through the tree and collects the polygon data from the visible leaves into a dynamic vertex and index buffer for rendering. Unfortunately, for large polygon datasets this technique is fairly useless. The memory copying of the vertex and index data into this dynamic buffer set will kill performance. An improvement to this method maintains a static vertex buffer and collects only indices during the traversal. This is a much better design, but still not quite optimal since memory copying is a costly operation no matter what, even if you are only dealing with 16-bit indices. We will examine an alternative approach in the next lesson that does not require a dynamic buffer (although they will be handy for specific tasks as we will discover later on).

Other strategies can also be employed to speed up the frustum rejection pass of the tree by reducing the number of plane/box tests that must be performed.

One of the most popular is called *frame coherence* (or sometimes *temporal coherence*). It works by having each node remember the first frustum plane that caused the node to be rejected so that in the next frame update, we can test the failed plane first and hopefully benefit from the fact that the node is still outside the frustum. This takes advantage of the fact that between any two given frames (a small time step), a node that was invisible last time is likely to be invisible again as a result of the same plane failure. The basic idea is that the node says, “The last time I was tested, I failed against frustum plane N, so let me check plane N first this time because it is likely that in the short amount of time that has elapsed, I will fail again against this plane and get rejected straight away”. This reduces the number of plane/box tests down from 6 to 1 when all goes well.

Just as we can reject large swaths of the tree when a node is outside the frustum, another important optimization can be implemented when a node’s bounding volume is found to be contained completely *inside* the frustum. If a node is totally inside the frustum then we know for certain that all of its child nodes must also be contained inside the frustum also. Thus, once we find a node fully contained in the frustum, we no longer have to test the bounding volumes of its children -- we can traverse immediately to its leaf nodes and render them. If we do not allow for node spanning polygons (i.e., they were clipped to the nodes during compilation), hardware clipping can also be disabled since we know that none of the polygons contained within the frustum will require clipping. This is not really a major savings these days since hardware clipping is quite fast, but it is worth noting nonetheless. If you ever need to implement a software renderer, this would be something to factor in.

Finally, another common optimization to reduce plane testing involves the child nodes benefiting from information that was learned during the parent node’s frustum tests. This works for cases where there was partial intersection. We know for example that if a node is found to be inside a frustum plane, then all of its children must also be inside that frustum plane. Therefore, there is no need to test that frustum plane against any of the child nodes. We can always assume that the child currently being tested against the frustum is inside that plane without performing any test.

All of the above techniques and optimizations will be fully explained and implemented in the next lesson when we discuss rendering our spatial trees in more detail. They have been introduced only briefly here to make you aware of the complexities of such a system and the various optimizations that can be performed.

14.2.3 Hierarchical Ray Intersection

One of the most common intersection routines used in games and related applications is the intersection test of a ray with a polygon. Such techniques are used to calculate light maps, form the narrow phase of our collision detection system, and determine whether line of sight exists between two objects in the game world. For example, we often wish to determine whether a ray cast from one location in the scene to another is free from obstruction (line of sight). The only way to know that without performing spatial partitioning is to intersection test the ray with every polygon in the scene. Only when we reach the end of the polygon list with no intersections found do we know that the ray is free from obstruction and a clear line of sight exists. Unfortunately, if our scene is comprised of many thousands of polygons, that query is going to be unsuitable for real-time applications. Furthermore, we may often want to perform such tests many times in a single frame update. Spatial partitioning speeds up such ray/scene intersection tests by reducing the per-polygon component. The only polygons that need to be individually tested against the ray are the ones inside the leaf nodes whose bounding boxes are intersected by the ray. To demonstrate this, in this next example we will assume that a scene has been spatially divided into a tree, resulting in 16 leaf nodes.

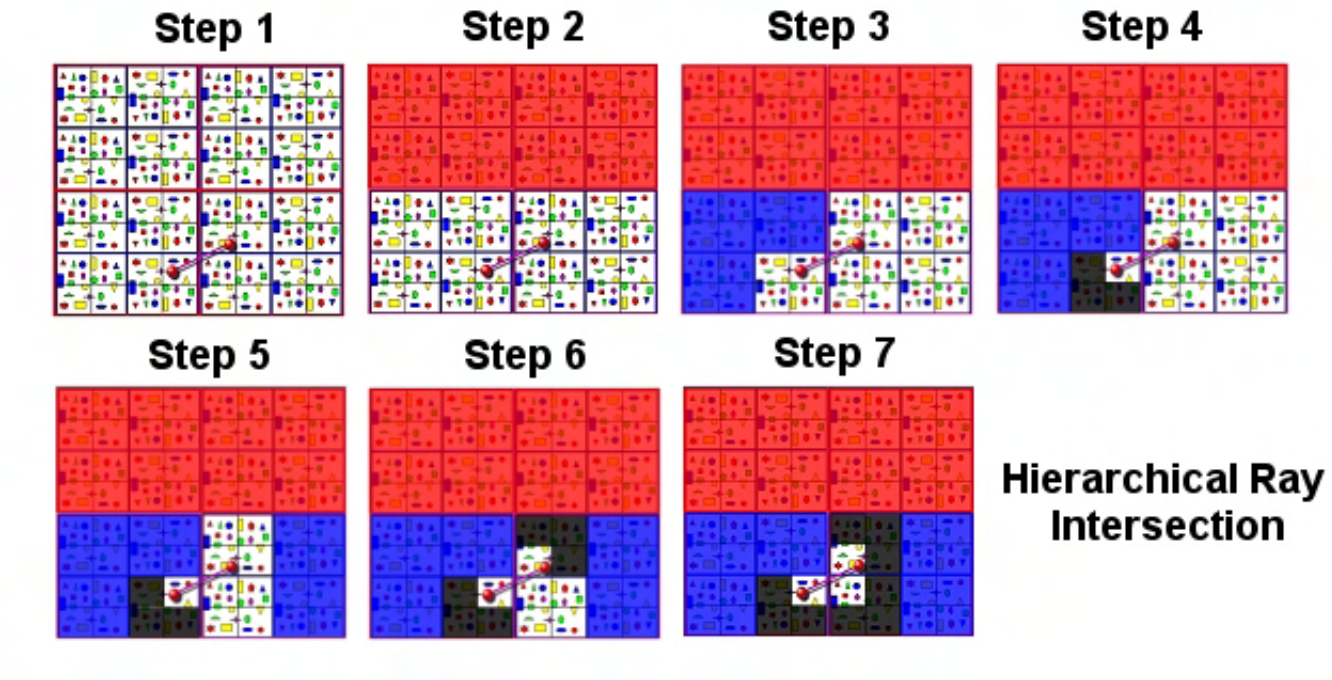


Figure 14.21

In Step 1 we see the ray end points (the two red spheres) and a purple line joining them. We can also see that the scene contains many polygons.

To find the polygons our ray intersects, we need to feed the ray through the tree and find which leaf nodes it eventually pops out in. We would start by feeding in the ray to the root node and testing it against the bounding volumes of each child of the root. As shown in Step 2, in this first stage, two of the root's children would be rejected because the ray does not intersect them. The rejected quadrants are highlighted in red.

Since we have found children of the root whose bounding volumes are intersected by the ray (the bottom two quadrants of the scene) we must send our ray into each child node. We will follow the path of the bottom left child node first. In Step 3 we can see that once in the bottom left quadrant of the root, we would then test the ray against the four child quadrants of this node. The ray is only found to be intersecting one of its children, so three are rejected from further consideration. These are highlighted in blue in the image.

From the blue children in the bottom left quadrant, only its bottom right quadrant node was intersecting the ray so we send it down to that node (Step 3). In Step 4 we show our ray visiting this node and being tested against that node's four children. Of its children, only one of them intersects the ray and three are rejected. The three rejected children are highlighted black in Step 4. At this point we send the ray into that child and find it is a leaf node which contains polygon data. We add the polygon data to a container so that we will have access to it when traversal is complete.

We have now traversed to the bottom of the tree entered via the bottom left child node of the root, so now it is time to send the ray down the root node's bottom right child. In Step 5 we see that as we visit the bottom right child of the root and test against its four children, two of them are not intersecting the ray (the blue ones). Two of the children are however, so we must traverse with our ray into each. First we start with the top left child where the ray is tested against its four child nodes and found to be contained in only one (Step 6). This one is also a leaf node, so the polygons are added to the polygon container. The rejected children are highlighted black. We have now finished with that branch of the tree so we step up a level and visit the other node our ray intersected -- the bottom left child of the bottom right child of the root (Step 7). Once again, we pass our ray into this node and test against its four children and find our ray is only in one of them. The ones that are rejected are highlighted in black. At this point we have reached the bottom of the tree, so we return. The recursive process unwinds right up to the root node and we have a container with polygons that were contained in the three intersected leaf nodes. These polygons can then be tested one at a time to see if an intersection really does occur.

In this simple example we have used a scene divided up into 16 leaf nodes so only $13/16^{th}$ of our polygon data would have to be tested at the per-polygon level. However, in a real situation we would have the scene divided up into many more leaves and the polygon data collected would be a mere fraction of the overall polygon count of the scene. To prove its efficiency, we can see that by performing four ray/box intersections tests at the root node we immediately reject half the total number of polygons in the scene regardless of the size of that scene.

Remember that although the spatial partitioning examples given here are partitioning polygon soups and storing individual polygons at the leaf nodes, this need not be the case. If your scene is represented as series of meshes for example, you could modify the node structure to contain a linked list of meshes instead of an array of polygons. Assigning a mesh to the tree would involve nothing more than sending its bounding volume down the tree and storing its pointer in the leaf nodes its bounding volume ends up

intersecting. You will often have a single mesh assigned to multiple leaf nodes, so you will have to make sure you do not render it twice during your render pass. In Lab Project 14.1 we will actually implement a spatial tree that manages both static polygon soups and dynamic meshes/actors. The static world space geometry loaded from the IWF file will be (optionally clipped and) assigned to the leaf node's polygon array. Each dynamic object (actor) will contain a list of leaves in which it is contained. When a dynamic object's position is updated, we will send its bounding volume through the tree and get back the list of leaves it intersected. We will store these leaf indices in the object structure. Before rendering any object, we will instruct the spatial tree to build a list of visible leaves. The object will ask the tree whether any of the leaves it is contained in are visible and if so, we can render the object. To be sure, there are other ways to manage this relationship, and we will talk more about it in the next lesson. For now, this gives you a fairly high level view of what is to come.

Now that we basically understand what spatial partitioning is, we will now look at some of the more common choices of spatial partitioning data structures (trees) that are used in commercial game development. There are many different types of trees used to spatially partition scenes into a hierarchy and the tree type you use for your application will very much depend on the scene itself and the type of partitioning it requires. In this lesson and the next we will discuss and implement quad-trees, oct-trees, and kD-trees. In Chapters 16 and 17 we will examine BSP trees. Keep in mind that there are a variety of different ways that each of these trees can be implemented. In this chapter we will discuss the vanilla approaches that tend to be the most common for each tree type. We will begin our exploration with quad-trees since we have already laid some foundation in the prior section.

14.3 Quad-Tree Theory

A quad-tree is essentially a two dimensional spatial partitioning data structure. This does not mean that it cannot be used to spatially partition three dimensional worlds, only that it spatially subdivides the world into bounding volumes along the (typically) XZ axis of the world. In a vanilla implementation where polygon data is only assigned to the leaf nodes, each one of these leaves contains all the geometry that falls within the X and Z extents of its bounding volume. No spatial subdivision is done along the third axis (traditionally, the Y axis). That is, the Y extents of every leaf node will be the same; the maximum and minimum extents of the entire scene, which creates bounding volumes with identical heights. As discussed in the last section, each node in the tree has four children that uniformly divide its space into quadrants.

Because of the two dimensional subdivision scheme used by the quad-tree, it is ideally suited for partitioning scenes that do not contain many polygons spread over a wide range of altitudes. For terrain partitioning, a quad-tree is a logical choice since a terrain's polygon data is usually aligned with the XZ plane. During a frustum culling pass, there will not be many times when parts of the terrain that are not visible will be situated above or below the frustum. Because every leaf node shares the same height, we will never be frustum culling geometry that is above or below the camera in a given location on the terrain. We might say that the quad-tree allows us to frustum cull data that is either in front or behind us and to the left or right of us. It is very unlikely that you will be standing on a terrain square and have parts of the terrain located nearby but very far above you or below you. Of course, there may be times

when you might be standing at the foot of a hill and perhaps the top of that hill could be culled because it is situated above the top frustum plane, but in general quad-trees work fairly well for terrains.

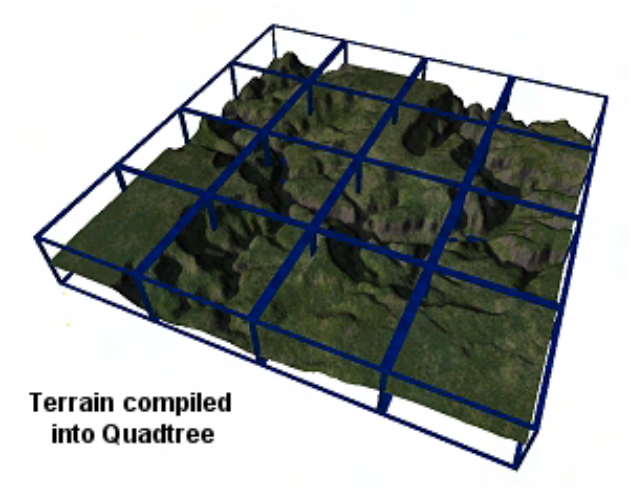


Figure 14.22

As you can see in Figure 14.22, a terrain like this has much larger dimensions along the X and Z axes of the world, so little would be gained from subdividing space vertically as well. This would introduce many more nodes into the tree and make traversals slower for very little (if any) benefit.

Figure 14.22 shows a terrain compiled into a quad-tree and although in reality this terrain would be compiled into many more leaf nodes than illustrated here, it should be obvious why this would be an efficient partitioning scheme for a terrain.

The height of each of the leaf node's bounding volumes can either be the max Y extents of the polygons that exist in that leaf node or it can be taken from the Y extents of the bounding box compiled for the entire scene at the root node. In the previous examples and in Figure 14.22, we have used the Y extents of the bounding box compiled for the entire scene for each leaf node's Y extents. We can see in Figure 14.22 that this generates nodes of identical height throughout the entire tree. In certain cases, leaf nodes may be more efficiently frustum culled or rejected from polygon queries earlier if the Y extents of each node's bounding box is not inherited from the bounding volume for entire scene but is instead calculated at node creation time from the polygon data in that node. Calculating the Y extents for a node would be easy -- we could just loop through each polygon that made it into that node during the compile and record the maximum and minimum Y coordinates of the polygon set. This will yield the minimum and maximum Y coordinates of any polygons stored at that node (or below it) which can then be used as the Y extents of its bounding box. This type of quad-tree (which we will refer to as a *Y-variant quad-tree*) generates children at each node which may not fill the space of the entire parent node, but will still always be totally contained inside it.

Figure 14.23 shows a how a quad-tree node normally has its space uniformly subdivided for each of its children. The height of all child nodes is equal to the height of the parent, even if there is no polygon data stored at that height. The circular inset shows how the children of the quad-tree node might look if, when each child node is created, the Y extents of its bounding volume are calculated from the polygon data that made it into that node. The Y-variant quad-tree is a favourite of ours here at the Game Institute as it performs consistently well in all of our benchmarks.

Before we discuss how to create a quad-tree, we should look at situations where it might not be the best choice of spatial manager due to its two dimensional spatial partitioning. A space-combat scene is one such scenario where the quad-tree might not be the best choice.

Figure 14.23

In such a scene, you could (for example) have a hundred complex spaceship meshes all situated at exactly the same X and Z coordinates in space but at different positions along the Y axis of the coordinate system. We can think of these spaceship models as being positioned in the world such that they would look like they were on top of each other when the scene was viewed from above. If we were using a quad-tree to spatially manage this scene of dynamic objects, all one hundred space ship models would exist in a single leaf node. When the camera entered this leaf node, all of the spaceships in this node would be rendered. This would be true even if all the ships were positioned far above and below the frustum. Essentially, we would be sending one hundred space ship models through the transformation pipeline even if none of them can be seen. Therefore in this example, where the scene has large Y extents, we would rather use a spatial partitioning technique that allows for the scene to be subdivided vertically as well. Although the examples given are associated with efficient frustum culling, the same is true for collision queries. If our swept sphere's bounding volume intersected that same leaf node, the polygons of all one hundred space ships would be collected by the broad phase and sent to the narrow phase as potential colliders (assuming we did no further bounding volume tests). However, the space ships could be positioned nowhere near the swept sphere.

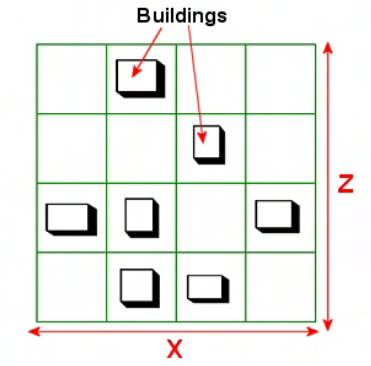


Figure 14.24

Another example where a quad-tree might not be the best choice is when representing a scene that models a cityscape with towering skyscrapers. We will use a very simple example to demonstrate why this is the case.

In the Figure 14.24 we see a scene consisting of seven tall buildings compiled into a quad-tree. In this first image we are looking at the scene from the top down perspective and at first glance this scene appears to fit nicely into a quad-tree.

In this example the scene has been divided into 16 leaf nodes along the X and Z axes of the world. What cannot be seen from this two dimensional representation is how high each building is, and thus how tall the bounding boxes are. In Figure 14.25 we see another view of the scene rendered three dimensionally to better demonstrate this point.

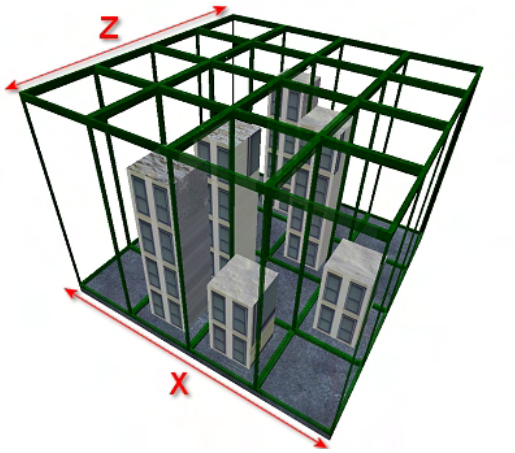


Figure 14.25

In this example, the Y extents of each quad-tree node are the same as the Y extents of the entire scene's bounding box, generating leaf nodes of identical size along the Y axis. Now let us imagine that the camera is located in front of the first two buildings at ground level with a rather narrow frustum.

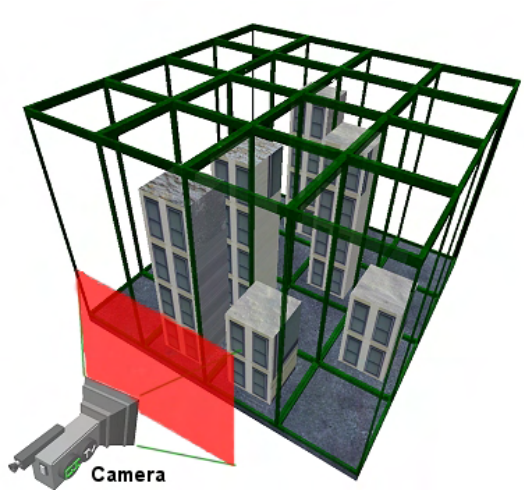


Figure 14.26

The red highlighted area in Figure 14.26 illustrates the section of the two front buildings that are actually inside the frustum and would need to be rendered. The camera can only see the bottom section of the two front buildings and therefore it is only the polygons comprising the bottom sections of each building that need to be rendered. However, because the polygons in each building are assigned to a single leaf node, when the leaf node is partially visible all of the polygons in that leaf are rendered. In this image we can see that the front three leaf nodes (with respect to the camera) can be partially seen from the camera position and as two of those leaf nodes contain buildings, these building will be rendered in their entirety. This is a shame as there are many polygons in each building which are situated well above the range of the frustum and they would be needlessly transformed and rendered (or collected as potential colliders). If you imagine a large cityscape scene consisting of hundreds of tall skyscrapers and imagine that the camera is usually situated at the ground level, using a quad-tree to partition the scene would result in many polygons in the upper regions of these buildings being rendered when they cannot even be seen. An oct-tree (discussed later) might prove to be a better choice of partitioning scheme in this instance.

You may be thinking that the simple answer is to always use an oct-tree instead of a quad-tree when working in three dimensions, but that is not a wise rule to live by. In our terrain example, nothing would be gained (or very little) by spatially subdividing the scene vertically. All we would achieve is a larger tree that is slower to traverse. More AABB/Frustum tests would need to be performed to reject a section of terrain that, in a quad-tree, would fit in a single leaf node and be rejected with a single AABB/Frustum test.

Note: Do not assume that an oct-tree is the best solution even if your scene does contain a large distribution of objects along the Y axis. Oct-trees create many more nodes which makes them slower to traverse and causes the polygons to be rendered in smaller batches. On modern hardware this might cause it to underperform with respect to the quad-tree. In most of our tests performed with the partitioning and rendering of static polygon data, the Y variant quad-tree outperformed the oct-tree in almost every case. However, this might not be true for a mesh based tree and would certainly not be true for our collision system's broad phase. We would not want to send polygon data to the expensive narrow phase which is above or below our swept sphere's volume. However, for rendering purposes, the quad-tree has been consistently hardware friendly. The lesson is to always to test the tree types on a variety of different scenes and machine configurations before deciding which one to use. Of course, you do not have to use the same tree for your collision geometry as you do for your render geometry. For example, you could compile your collision data using an oct-tree but render the scene data using a quad-tree.

14.3.1 Partitioning a Quad-Tree Node

When building a quad-tree node, we will essentially have a list of polygons that are contained in the parent node volume that need to be assigned to its four child nodes. The parent volume will be subdivided into four equally sized volumes along the X and Z axes. The centerpoint of the parent node's bounding box describes the position where the edges of each of the four child node's bounding boxes will meet. Each child node is created and is assigned a bounding volume representing a quadrant of the parent node (top left, top right, bottom left or bottom right). Once we have the child nodes and their bounding volumes calculated, we can start to test which child node(s) a given polygon in the parent node's polygon list is in. It is possible that a polygon in the list passed down from the parent might span the bounding volumes of multiple child nodes. In such a case we can either choose to add the polygon to the polygon list of each child it intersects, or we can clip the polygon to the bounding volumes and store each polygon fragment in the child for whose bounding volume it will now be totally contained.

Placing the polygon in multiple leaf nodes does come with its fair share of problems. We will need to make sure that we do not render or query the polygon multiple times if more than one of the leaves in which it exists is being rendered or queried. This can add some traversal overhead, although not much. Another problem with not clipping the polygons is that when a leaf node is rendered or queried, the node no longer describes the polygons exactly stored in that volume. For example, we might assign a polygon to a leaf node that is much larger than the node. When that node is visible, we have a looser fit for the exact data that can be seen and should be processed. Of course, the problem with clipping is that we essentially perform a process that creates two polygons from one. If this happens hundreds of times during the building procedure it is not uncommon for the polygon count to grow between 50 to 90 percent depending on the type of tree you are using and the size of the leaf nodes. The larger the leaf nodes, the less clipping will occur but more polygons will be contained in a single leaf.

In our implementation, we will provide options for clipping the static polygon data to the bounding volumes of the tree and will also implement a system that elegantly handles the sharing of polygons between multiple nodes when clipping is not being used.

We will store in our node its bounding volume and two clip planes that are used for the classification and clipping of polygon data during the tree building process. We will discuss how to clip polygons to planes later in this lesson. The benefit of clipping is that any node in the tree will contain an exact fit of

the data contained inside it and more importantly, no polygon will ever belong to multiple leaf nodes. Of course, it does mean that whenever a split happens we introduce more polygons into the scene.

> **Note:** Until otherwise stated, we will assume for the sake of the next discussion that we are clipping our polygon data to the nodes in which they belong. Later we will discuss the non-clip option.

Please note that in this section we are currently talking about the assigning of static world space polygon data to the tree, such as the internal geometry loaded from a GILES™ created IWF file. As this data never changes throughout the life of the application, we will clip it exactly to the tree at compile time and store the polygon data in the leaves. Later we will discuss how we can also link dynamic objects to leaf nodes in the tree. These objects are not clipped and are linked to the leaf nodes implicitly by storing a list of leaves they are contained within. Meshes and actors do not have their polygon data stored in the tree. Instead we will maintain an internal array of leaf indices in which the object currently resides. The spatial tree will expose a method that will allow dynamic objects to pass their bounding volume down the tree and get back a list of leaf indices in which they are currently contained. The dynamic object can optionally store these indices and only render itself if any of these leaves are visible (although the system will maintain this information internally for later querying if desired).

Because a given node is really just an axis aligned bounding box, generating the clip planes for a quad-tree node is delightfully easy. Two planes will be needed: one that will split the node's volume halfway along its depth (Z) axis and another that will split the volume half way along its width (X) axis. We know that the normal to a plane that will divide its depth in two will be <0, 0, 1> (or <0, 0, -1> as it represents the same clip plane) and that the centerpoint of the node's bounding volume will be a point on that plane as shown in Figure 14.27.

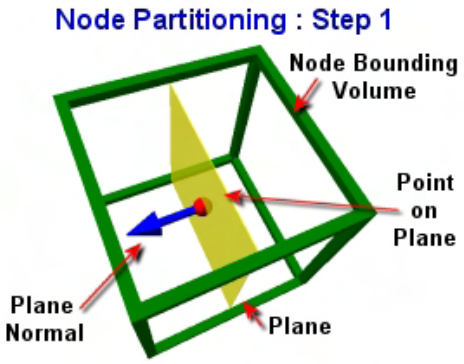


Figure 14.27

In this image we are assuming that the blue arrow is the plane normal pointing along the world Z axis and that the red sphere is the centerpoint of the node's bounding volume. Thus, these two pieces of information are all we need to describe the clip plane shown as the yellow slab in Figure 14.27.

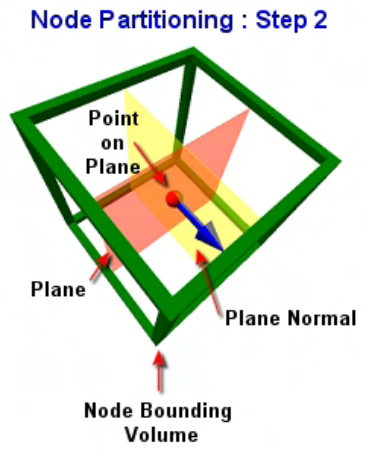


Figure 14.28

The creation of the second clip plane is equally as easy. Once again, the centerpoint of the bounding volume describes the point on the plane and this time we use a normal of <1, 0, 0> so that we have a plane that divides the volume's width in two.

Once we have generated the two planes, we take each polygon in the node's list and clip it to each plane. This is done in two passes. For each polygon in the list we classify it against the first node plane. If it is either in front or behind the node we leave it in the list and continue on to the next polygon. If we find that a polygon is spanning the plane we split it at the plane. The original polygon in the list is deleted and the two new split polygons are added to the list. After every polygon in the list for this node has been tested against the first plane, any polygon that was spanning the plane will have been removed from the list and replaced with two polygons that fit entirely in the front and back half space of the node's volume.

We then clip this list against the second plane using the same scheme. That is, each polygon is classified against the plane and if not spanning, the second plane is left in the list unaltered. If it does span the second plane, we split the polygon in two, deleting the original from the list and adding the two split fragments in its place. At the end of this process, we will have not yet determined which child nodes each polygon should be assigned to, but we know the polygons have been clipped such that every polygon will neatly fit into exactly one child node (one quadrant).

The following code shows this step. Do not worry too much about how the actual functions that are called work at the moment; we will get to all that later when we implement everything. For now just know that this code would be executed during the building of a node. The node is passed an STL vector of all the polygons that have made it into that node during the compilation process so far. For the root node, this vector will contain all the polygons in the scene. Each polygon is assumed to be represented by a CPolygon class that has a method called 'Classify' which returns a flag describing whether it is in front, behind, or spanning a plane. It also has a method called 'Split' which splits the polygon to a plane and returns two new CPolygon structures containing the split polygon fragments. Remember, this code is not yet trying to determine which polygon in the list should be passed to which child node, it is just clipping any polygons that straddle the quadrant borders.

The first thing we do is generate the two clip planes using the D3DXPlaneFromPointNormal method. This method accepts as its parameters a point on the plane and a normal and returns (via the first parameter) the plane represented as a D3DXPlane structure (in a,b,c,d format). For both planes, the point on plane is simply the centerpoint of the node's bounding box (2nd parameter). For the plane normal of the first plane we pass in the vector (0,0,1) which is the world Z axis; for the second plane we pass the vector (1,0,0) which is the world X axis.

```
D3DXPlane ClipPlanes[2];

D3DXPlaneFromPointNormal( &ClipPlanes[0], &((BoundsMin + BoundsMax) / 2.0f),
                          &D3DXVECTOR3( 0.0f, 0.0f, 1.0f ) );

D3DXPlaneFromPointNormal(  &ClipPlanes[1], &((BoundsMin + BoundsMax) / 2.0f),
                           &D3DXVECTOR3( 1.0f, 0.0f, 0.0f ) );
```

Notice in the above code how we have allocated an array of two D3DXPlane structures on the stack which will receive the clip planes that we calculate. In the first element we store the XY plane and in the second element we store the YZ plane.

Now that we have both planes temporarily calculated for the current node, we will classify the polygons in the list against each one. Therefore, we set up an outer loop for each plane and an inner loop that tests each polygon against the current plane being processed.

```
    // Split all polygons against both planes
    for ( i = 0; i < 2; ++i )
    {
        for(PolyIterator= PolyList.begin();PolyIterator != PolyList.end(); ++PolyIterator)
        {
            // Store current poly
            CurrentPoly = *PolyIterator;
            if ( !CurrentPoly ) continue;
```

The first thing we do is test the current polygon being processed to see if its pointer is NULL, and if so, we skip it. You will see in a moment why we perform this test.

Now that we have a pointer to the current polygon we want to test we will call its Classify method and pass in the current plane. The Classify method of CPolygon simply returns a value that describes the position of the polygon with respect to the plane. The four values it can return are CLASSIFY_INFRONT, CLASSIFY_BEHIND, CLASSIFY_ONPLANE and CLASSIFY_SPANNING. As you can see, these tell us whether the polygon we are testing is in front or behind the plane or whether it is spanning the plane. In this loop, we are looking for polygons that are spanning the current plane being tested because if we find a polygon spanning a plane, we must spilt it into two new polygons at that plane. Here is the remainder of the code.

```
            // Classify the poly against the first plane
            ULONG Location = CurrentPoly->Classify( ClipPlanes[i] );

            if ( Location == CPolygon::CLASSIFY_SPANNING )
            {
              // Split the current poly against the plane,
              // delete it and set it to NULL in the list
                CurrentPoly->Split( ClipPlanes[i], &FrontSplit, &BackSplit );

                delete CurrentPoly;
                *PolyIterator = NULL;

                // Add these to the end of the current poly list
                PolyList.push_back( FrontSplit );
                PolyList.push_back( BackSplit );

            } // End if Spanning

        } // Next Polygon

    } // Next Plane
```

As you can see, if the classification of the current polygon does not return CLASSIFY_SPANNING then we leave it in the list and skip on to the next one. That does not mean of course that this same polygon will not be clipped when the second plane is tested in the second iteration of the outer loop. If

the polygon is spanning the current plane then we call its Split routine. This function takes three parameters. The first is the plane we would like to clip the polygon against and the second and third parameters are where we pass in the address of CPolygon pointers which on function return will point to the new split fragments.

After this function returns, FrontSplit and BackSplit will contain the two polygon fragments that lay in front of the plane and behind it, respectively. At this point we no longer want the original polygon in the list as it will now be replaced by these two fragments. Therefore, we delete the original CPolygon structure and set that element in the STL vector to NULL. We then add the front and back splits to the polygon list. Of course, these split fragments may each get clipped again when they are tested against the second plane in the second iteration of the outer loop.

Notice in the above code that whenever we split a polygon, that original polygon is deleted and its pointer in the STL vector is replaced with a NULL. The two split polygons are added to the end of the vector. Now you can see why we did the polygon pointer test for NULL at the top of the list. When processing the second clip plane, many of the pointers in the list may be NULL. There will be a NULL in the list for every original polygon that was split by the first plane.

Now that we have the polygons such that every polygon is contained in exactly one quadrant of the node, it is now time to build four children polygon lists. That is, we need to calculate which polygons will need to be passed into each child node. We will build a list for each quadrant which will result in four polygon lists which we can then pass into the child nodes during the recursive build process. The next section of code shows these four polygon lists (STL vectors) being compiled.

```
PolygonList            ChildList[4];

// Classify the polygons and sort them into the four child lists.
for(PolyIterator = PolyList.begin(); PolyIterator!=PolyList.end(); ++PolyIterator )
{
 // Store current poly
 CurrentPoly = *PolyIterator;
 if ( !CurrentPoly ) continue;

 // Classify the poly against the planes
 Location0 = CurrentPoly->Classify( ClipPlanes[0] );
 Location1 = CurrentPoly->Classify( ClipPlanes[1] );

 // Position relative to XY plane
 if ( Location0 == CPolygon::CLASSIFY_BEHIND )
 {
    // Position relative to ZY Plane
    if ( Location1 == CPolygon::CLASSIFY_BEHIND )
         ChildList[0].push_back( CurrentPoly );
    else
         ChildList[1].push_back( CurrentPoly );

  } // End if behind

 else
  {
    // Position relative to ZY Plane
```

```
    if ( Location1 == CPolygon::CLASSIFY_BEHIND )
         ChildList[2].push_back( CurrentPoly );
    else
         ChildList[3].push_back( CurrentPoly );

  } // End if in-front or on-plane

} // Next Polygon
```

The code first allocates four empty CPolygon vectors that will contain the polygon lists for each child node that we generate. We then loop through each polygon in the clipped list that we just modified. Remembering to skip past any polygon pointer that is set to NULL in the array, we then classify the polygon against both of the node's clip planes and store the results in the Location0 and Location1 local variables. Finding out which list the polygon should be assigned to is now a simple case of testing these results.

We can see in the above code that if the polygon is found to be behind the first clip plane (the Z axis split) then the polygon is obviously contained in the back half space. We then test to see what its classification against the second clip plane was. Remember the second clip plane is the plane that splits the width of the volume and has a normal (1,0,0). If it is behind then we know that not only is the polygon in the back halfspace of the node, but it is also in the left halfspace behind the first clip plane. Thus, this polygon must be contained in the top left quadrant. The else case says that if we are not behind the second clip plane but we are behind the first clip plane, then this polygon must be contained in the top right quadrant of the node. We perform the same tests when the polygon is found to be in front of the first clip plane to determine whether it belongs to the bottom left or bottom right quadrant.

As the above code shows, once we find the quadrant a polygon is in, we add it to the relevant polygon list. After this code has executed, ChildList[0] will contain the polygons for the top left quadrant and ChildList[1] will contain the polygons for the top right. ChildList[2] and ChildList[3] will contain the polygons for the bottom left and bottom right quadrants, respectively.

At this point, all that would be left to do is allocate the four child nodes, construct their bounding boxes as quadrants of the parent node's volume and recur into each child sending the list that was compiled for it by the parent node and the whole process repeats. Only when the list of polygons is small or the bounding box of the child node is small do we decide to stop subdividing and just assign the polygon list to the node, making it a leaf.

> **Note**: Do not worry if the above code did not provide enough insight into how to fully create a quad-tree. It was intended only to show the clipping that happens at each node during the build process. Later in this lesson we will walk through the code to a quad-tree compiler line by line.

Notice that the clip planes will be generated only during the construction of the node's child lists and they will not be stored. The only thing we store in the node (apart from any polygons) is its axis aligned bounding box and the child node pointers. Of course, you could decide the store the clip planes in the node as well if you think you will need them at a later stage. It is possible that you may wish to perform some query routines on the tree using the clip planes instead of the axis aligned bounding boxes, but we do not in our lab project. However, we will store the clip planes in the nodes of the kD-tree that we implement (and in the BSP tree, as we will see later in the course).

We now have a good idea of exactly what a quad-tree is, and this will go along way towards our understanding of the other spatial tree types. Later in this chapter we will cover the source code to a quad-tree compiler which we can then use in our applications. Each tree type we develop will all be derived from an abstract base interface, so we will be able to plug any of them in as the broad phase of our collision system.

That concludes our coverage of the quad-tree from a theoretical perspective, so we will now go on to discuss the other tree types. After we have discussed the various tree types, we will examine the separate processes involved in building them, such as the clipping of polygons and the mending of T-junctions introduced in the clipping phase.

14.4 Oct-Tree Theory

The great thing about the topic of hierarchical spatial partitioning is that whether we are implementing a quad-tree, an oct-tree or a kD-tree, the building and traversal of those trees are almost identical in every case. They are all trees consisting of nodes which themselves have child nodes which can be queried for intersection and traversed recursively. This is especially true in the case of an oct-tree where the only real difference between the quad-tree and the oct-tree is in the number of children spawned from each node.

In a quad-tree, each node's bounding volume is divided into quadrants along the X and Z axes and assigned to each of its four children. In an oct-tree, a node's bounding volume is divided along the X and Z axes and also along the Y axis. Therefore, each non-leaf node has its bounding volume divided into octants ($1/8^{th}$) and each non-leaf node of an oct-tree has eight children instead of four. The bounding volume of each child node represents $1/8^{th}$ of the parent node's bounding volume. Figure 14.30 shows how a single node's bounding box is divided into octants within an oct-tree, compared to being divided into quadrants for a quad-tree.

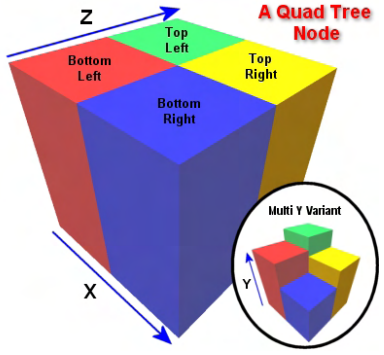


Figure 14.29

Figure 14.29 shows how the bounding volume of a non-terminal quad-tree node is subdivided into four smaller bounding volumes along its X and Z axes. In this image the child nodes each have the same height taken from the Y extents of the entire scene (the root node's bounding box). The circular inset in the image shows a variation of the quad-tree where the Y extents of each child node are calculated using the actual polygon data that was passed into that node during the building process.

In Figure 14.30 we see how the bounding volume of a non-terminal oct-tree node is divided into octants. Not only do we divide the parent node's bounding volume in two along the X and Z axes as we do with the quad-tree, but we also divide the parent volume into two along the Y axis. This creates eight child bounding volumes instead of four. The subdivision of the node's bounding volume resembles a double decked version of the quad-tree subdivision. Instead of just having Top Left, Top Right, Bottom Left and Bottom Right child nodes, the child nodes can now be described by prefixing the label with the deck to which they belong: Upper Top Left, Upper Top Right, Upper Bottom Left, Upper Bottom Right, Lower Top Left, Lower Top Right, Lower Bottom Left, and Lower Bottom Right as labeled in the diagram

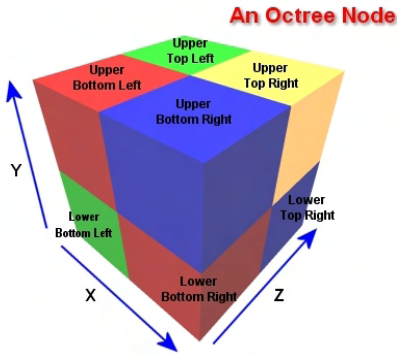


Figure 14.30

As you might imagine, building an oct-tree is a nearly identical process to building a quad-tree. The exception being of course that each non-leaf node has eight children instead of four so we have to divide our polygons into eight bounding boxes instead of four when building each node. Everything else is as before. The leaf nodes are the nodes at the end of a branch of the tree and in our implementation, are the only nodes that can contain geometry data.

The process of building an oct-tree requires only small changes to the code we saw earlier. The node structure used for an oct-tree will have pointers to eight child nodes instead of four and when building the node itself (in our implementation) we will now use three clip planes instead of two to build the clipped polygon lists for each child (see Figure 14.31).

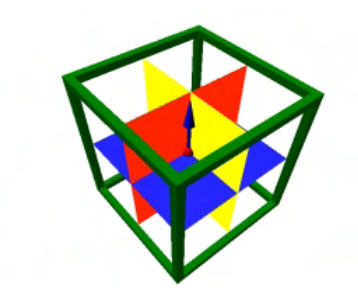


Figure 14.31

Trying to draw an image of what an oct-tree looks like in memory on a piece of paper is virtually impossible if the oct-tree is more than a few levels deep. Each node has eight children, each of which have eight children of their own, and so on right down the tree. In fact, if we think about how many leaves a 10 level oct-tree would partition our scene into, we would get a staggering result of 8^10 = 1,073,741,824 nodes at the lowest level of the tree. Not surprisingly, it is pretty much never the case that we will compile an oct-tree that has anywhere near this many levels. The good thing about oct-trees then is that they are typically going to be fairly shallow tree structures. Of course, when traversing the tree, we have eight child tests to perform at each node (instead of just four in the quad-tree case) to find the child nodes we wish to step into.

Figure 14.32 depicts a partial oct-tree that is three levels deep. The gray node represents the root node of the tree, the red nodes represented the eight immediate child nodes of the root, and if the width of a printed page was not an issue, each one of these nodes would have their own eight blue child nodes which in this example are leaf nodes. So that we can fit the image on a piece of paper we have only shown the child nodes of two of the red nodes. Remember, each one of these red nodes would have their own eight blue leaf nodes. Hopefully, this diagram will illustrate just how much the quad-tree and the

oct-tree are alike at their core. The oct-tree has double the number of branches leaving each node obviously, but it has the same arrangement with leaf nodes at the branch tips.

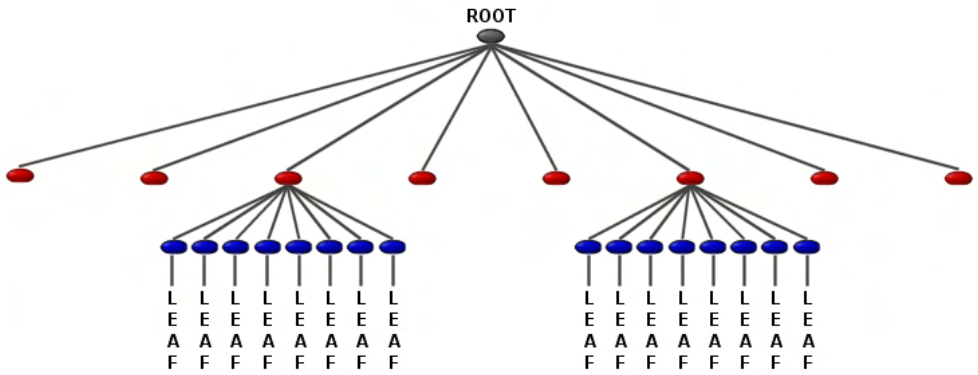


Figure 14.32

Rendering the oct-tree would also not be that different from the quad-tree case. The only real difference would be that when traversing the nodes of an oct-tree, we now have to perform Frustum/AABB tests against eight bounding volumes at each node instead of four, before we traverse down into the visible children.

As you can see from examining Figure 14.33, with an oct-tree, the scene is also divided vertically as well. Looking at the same position and orientation of the camera in this image, we can see that only the bottom two leaf nodes fall within the camera's frustum so only the polygons assigned to those leaf nodes would need to be rendered, which in this case would be the polygons in the bottom sections of each building only. The only time the top sections of the buildings would be rendered is when the camera is rotated upwards (like a person looking up at the sky) because only then would the upper leaf nodes intersect the frustum. At this point however, there is a good chance that the bottom leaf nodes of the building would no longer be inside the frustum and therefore when rendering the upper portions of the building in this example, the lower portions of each building will be frustum rejected and not rendered. Clearly this shows the advantages of an oct-tree and the limitations of the quad-tree in certain scenarios. We have seen that even if the camera is not rotated upwards, using a quad-tree, the entire building would be rendered even if the upper portions of the building cannot be seen.

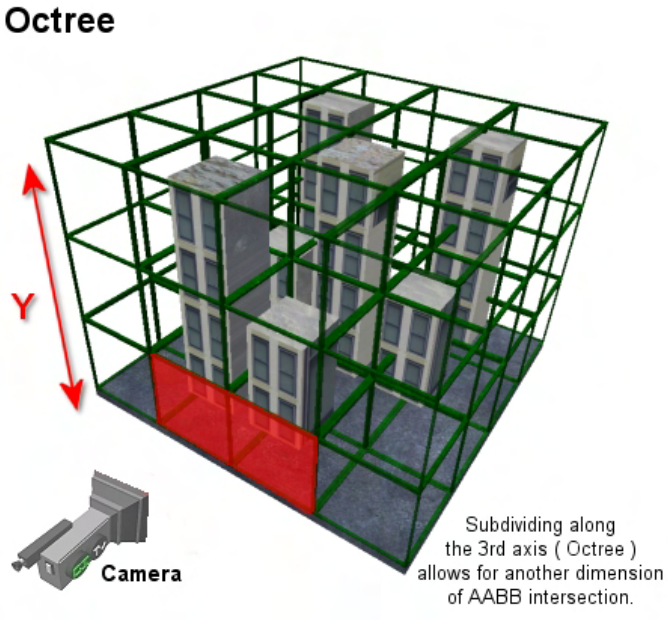


Figure 14.33

Of course, the same would also be true of intersection queries, if an object is contained in the lower leaves, only the polygons comprising the base of the buildings would need to be tested in the narrow phase. So we have discovered that the oct-tree allows us to more finely collect or cull the number of

polygons that need to be considered. If the scene geometry is distributed over a large vertical range, the oct-tree is often a better choice for multi-level indoor environments, cityscapes and areas that have immense freedom of movement along all three axes (such as a space scene). That being said, you should always benchmark your trees and find out for sure which is the best performer for a given scene. As mentioned previously, the oct-tree does suffer from creating more nodes to traverse and clips the polygon data much more aggressively. If care is not taken, you could find your polygon count doubling during oct-tree compilation.

14.5 kD-Tree Theory

A kD-tree is a partitioning technique that partitions space into two child volumes at each node. Each node contains a single split plane and pointers to two child nodes. The split planes chosen at each node are always axis aligned to the world and alternate with tree depth to carve the world into rectangular regions, much like an oct-tree. That is, at the first level of the tree, a split plane that partitions the space along the X axis might be chosen to create two child volumes. When each child is partitioned, the split plane used would divide their space along the Y axis to create two child volumes for each. For each of their children, the split plane used would be one that divides their volume along the Z axis. For their children, the process wraps around to the beginning and we start using the plane that divides space along the X axis again. This process of alternating between three axis aligned clip planes at each node repeats as we step down the tree until we wish to stop our subdivision. That node is then considered a leaf node.

Because the kD-tree partitions space in two at each node, it is a binary tree. Further, because this particular binary tree is used to partition **space** at each node, we might also refer to it as a binary space partitioning tree (BSP tree). While this is certainly true, the kD-tree is not normally what people are referring to when a BSP tree is referenced. A BSP tree is almost identical to a kD-tree except for the fact that at every node an arbitrarily oriented plane can be chosen (instead if using an axis-aligned one). This allows the BSP tree to expose very useful properties that allow for the determination of what is solid and empty space within the game world, which can then be used to optimize rendering by an order of magnitude. Because the kD-tree uses axis aligned planes, a kD-tree is often referred to as an *axis aligned BSP tree*.

The kD-tree allows space to be partitioned arbitrarily at each node as long as the clip plane is still being aligned to a world axis. That is, clip planes can exist at variable positions within the node's volume. The clip plane does not always have to divide the node's volume into two uniformly sized child volumes. This allows for the space to be subdivided such that only areas of interest (where geometry exists) get divided (more on this in a moment).

To summarize, a kD-tree is a spatial partitioning tree with the following properties.

- Each node represents a rectangular bounding volume. Just like an oct-tree or a quad-tree, the faces of each node's bounding box are aligned with the axes of the coordinate system.
- Each node stores a single split plane which is aligned to one of the coordinate system axes. This split plane does not have to cut the region into two equally child volumes (although we may wish it too).

- Each node has two child nodes representing the volumes each side of its split plane.
- The world axis used for the alignment of a node's split plane alternates with tree depth.

Building the kD-tree is much the same process as for an oct-tree or a quad-tree although there may be some differences in how the split plane at the node is chosen. To get across the basic idea of a kD-tree we will show images to illustrate the subdivision of a scene during the construction of such a tree. In this example we will simply calculate each node's split plane to cut through the center of the node's bounding volume. This will divide each node's volume into two equally sized child volumes. As you will see in the following examples, because the split plane being chosen partitions each node's volume into two along its center, we partition the scene in the same way as an oct-tree.

In this first image we show the construction of the root node. After creating the node we would loop through each of the scene's polygons and compile a bounding box that encompasses them all. This will be assigned to the root node as its bounding volume. At this point we have the root node's volume and a list of polygons that need to be passed into two child nodes. Our next task it to choose a split plane for the root. In this image we create a split plane whose normal is aligned with the world X axis (1,0,0). In this example we are using the center of the bounding volume as the point which describes the plane during plane creation and as such the plane cuts through the center of the box dividing its width in two.

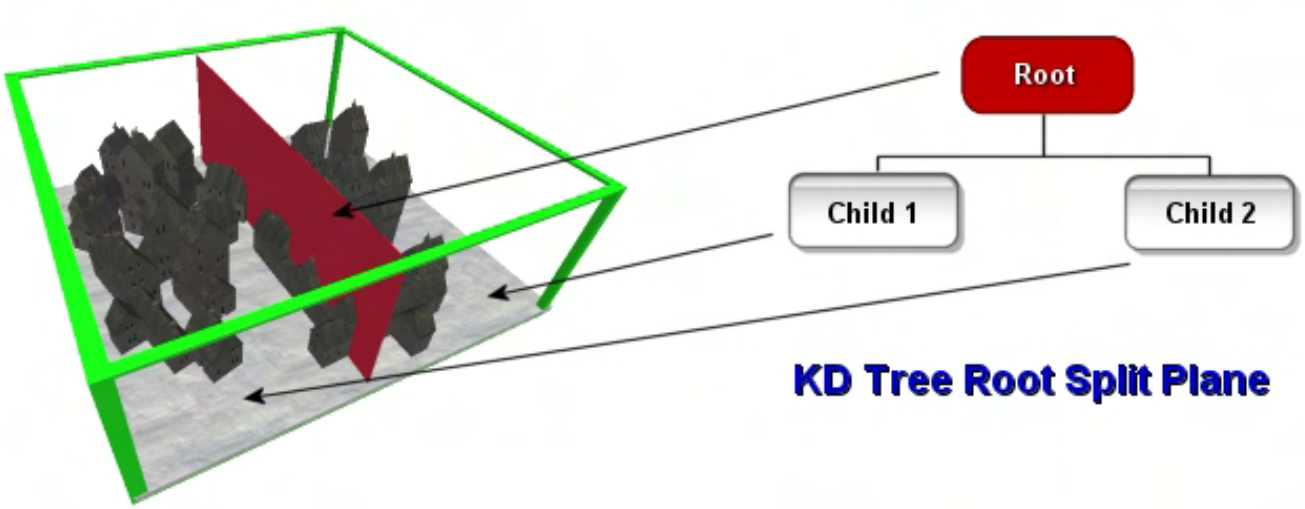


Figure 14.34

At this point we store the split plane in the node and loop through each polygon in the list and classify them against this plane. This allows us to build two polygon lists (one for each child node) describing the polygons that belong in each list. We will assume for this demonstration that we are once again clipping our geometry to the partitions, so any polygon in the root list that straddles the plane will be clipped in two by that plane and the each child fragment added to the respective child list.

At this point we have to recur into each of the children and perform exactly the same task (just as in the oct-tree/quad-tree case). The difference being now that as we step down to each level of the tree, we alternate the normal that we are going to use for the node's split plane. In Figure 14.34 we can see that at the first level in the tree we used a plane normal which is equal to the X axis of the coordinate system. This cuts the box's width in two.

In the next level of the tree we have to switch the clip plane to use one that is aligned to the Z axis of the system (0,0,1). That is, every node at the second level of the tree will use this same normal for its clip planes, as shown in Figure 14.35.

> **Note:** It does not really matter the order in which you alternate the clip planes used at each level of the tree, as long as you alternate between the three as you step through the levels of the tree. That is, you could use a plane normal equal to the world Y axis at the root, one that is equal to the world Z axis at the second level and one that is equal to the world X axis at the third level. As long as you repeat the pattern, everything will be fine and you will have a valid kD-tree.

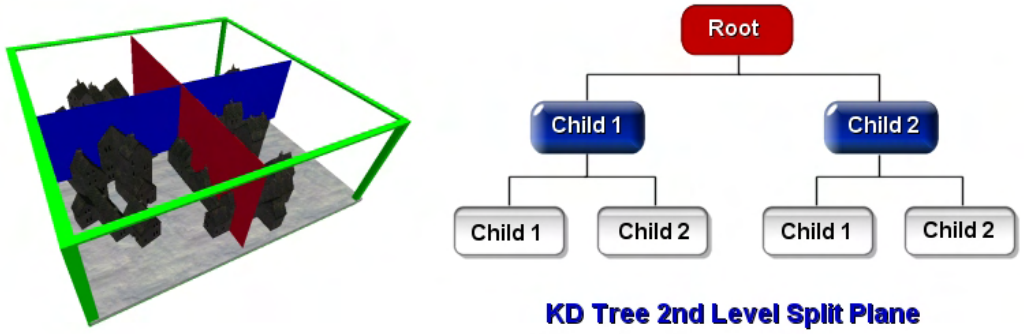


Figure 14.35

Figure 14.35 illustrates what happens when we step into each blue child of the root. For each child we are passed a list of polygons that are known to fit inside this child. We then compile a bounding box for that child based on its list of polygons and store it in the node. A split plane is then chosen which divides the child volume in two. Because we have stepped down a level, we alternate the world axis we use as our plane normal. In this example, the center point of the box is still used to construct the plane, which equates to a plane that splits each child into two equally sized children. In Figure 14.35 a plane aligned with the XY plane (the blue plane) of the coordinate system is being used for each child node, which carves each of their volumes along the depth coordinate. The polygons passed into each child node are then classified against (and clipped to) the split plane to build two child lists for each of its children (the white nodes in the hierarchy diagram).

In Figure 14.36 we see the construction of the next level of the tree, where the children of each child of the root are constructed.

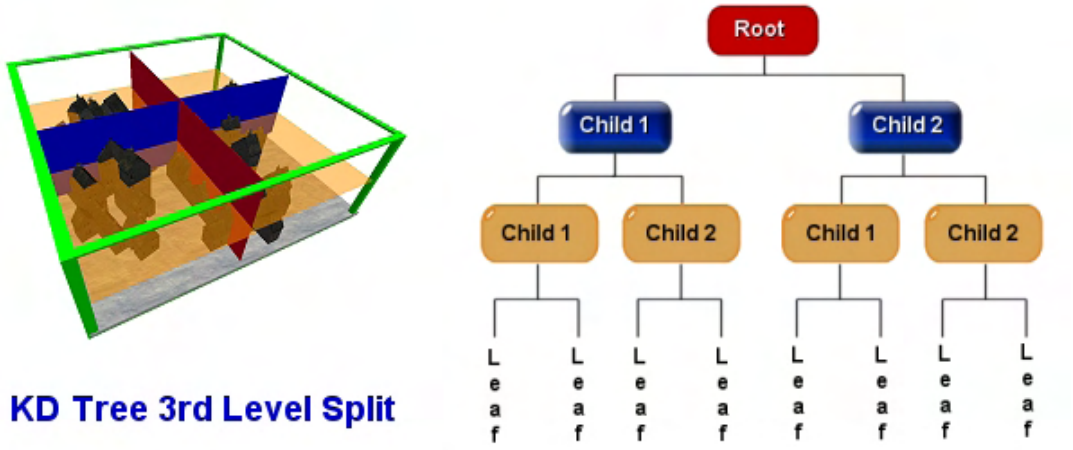


Figure 14.36

In Figure 14.36, when we step into the third level of the tree, the split plane alternates again. This time a plane normal aligned with the Y axis of the coordinate system is chosen. This creates a plane that carves each orange node in two vertically. Once again, the polygons that made it into each orange node would be classified against its plane to create two polygon lists for each of its children. In this diagram, we are assuming that when we stepped down to the fourth level of the tree, the child node's polygon list was considered to be so small that it was not worth subdividing further. At this point, we create leaf nodes which contain the actual polygon data. A leaf node is just a node like any other, with respect to representing an area of space. It has a bounding volume, no children, and it has polygon or mesh data associated with it.

We will see later that building and querying a kD-tree is even easier than both the oct-tree and the quad-tree since we have only one plane and two children to worry about at each node and we are simply trying to determine in which of the two children our query volume belongs. Performing a ray intersection test on a kD-tree is extremely simple since we can essentially just perform a ray/plane intersection test at each node.

Although the splitting strategy used above carves the space up in a uniform way like an oct-tree, this need not be the case for the kD-tree. At any given node we must always choose the correct axis aligned plane to split with, but that plane need not cut the volume into two equally sized child volumes. This is where a kD-tree generalizes the oct-tree, by allowing the world to be carved up into arbitrarily sized AABBs.

Figure 14.37 shows the same tree but with varying split plane positions being used for each child of a node. We can see that the root node (the red plane) splits its volume into two equal volumes as before. Both of its children split their volumes (the blue planes) at arbitrary positions along the depth of their volumes. We can also see that all the children at the 3rd level of the tree (the orange planes) have split planes at different heights from one another.

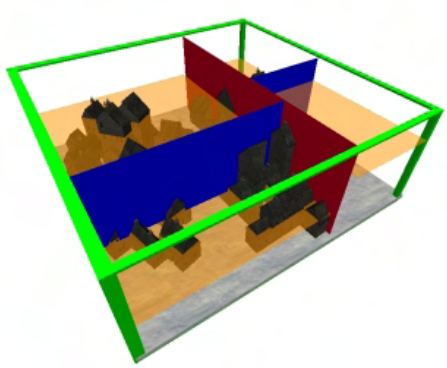


Figure 14.37

A useful characteristic of the kD-tree is that it allows us to favor subdivision in areas of the level that are more densely populated with geometry while still maintaining a balanced tree.

We now have a basic understanding of what a quad-tree, oct-tree, and kD-tree is, even if we are not yet intimate with the concepts behind coding each of these tree types. We are almost ready to start studying the implementation of our tree system, but before we do we will discuss the importance of tree balance, how to clip and split polygons to nodes, and explore exactly what T-junctions are and how we can repair them. Once done, we will be ready to start examining the code to all our tree types.

14.6 Mesh Trees and Polygon Trees

Whether you decide to store raw polygon data at the leaves of a spatial tree or just store whole meshes (or even clusters of meshes) is a decision you will have to make based on what the needs of your application are. Let us first have a quick discussion of the options.

14.6.1 Polygon Trees (Clipped/Unclipped)

If the geometry you intend to store in your tree is a totally static scene (e.g., a large static indoor level), then subdividing the scene at the polygon level might prove advantageous. This is especially true when using the tree in the broad phase of a collision system. A single leaf node will more accurately describe only the polygons that are contained within it, providing the ability to reject trivial data during polygon queries. A leaf node will contain only the polygons that are stored within the bounds of that leaf and the only polygons you are interested in intersection testing if the swept sphere is contained inside that leaf during a collision test. For rendering purposes, when a leaf is inside the frustum, the data we render associated with that leaf contains very few potentially non-visible polygons, even if the polygon data has not been clipped to the leaf nodes.

Obviously, if the polygon data has been clipped to the planes of the leaf node, then a leaf node will always contain an exact set of polygon data that fits inside that leaf node. Building the tree at the polygon level is obviously much slower than at the mesh level because every individual polygon has to be sent down the tree until it pops out in a leaf node and is added to its polygon buffer. If clipping is

being used then this process is slower again, and more polygons will be produced as a result. Certainly, the per-polygon approach is not ideal if the scene is largely dynamic because whenever geometry is moved, the tree will need to be rebuilt from scratch. This could potentially take an unacceptably long time and will not be practical to do in real-time between frame updates. If the scene is static however, then this is not a concern and we can benefit from the more exact fitting of the polygon dataset.

14.6.2 Mesh Trees

When building trees at the mesh level, tree construction is much quicker. We are no longer concerned with the individual polygons that a mesh is constructed from, but instead, we just use its world space bounding volume to determine which leaf nodes the mesh belongs in. This is extremely quick to do because even if a mesh contained 10,000 polygons, when traversing the tree to determine which leaf nodes the mesh belongs in, we are simply doing an AABB/AABB test at each node until we locate all the leaves the AABB intersects. The bounding volume test we perform when classifying the mesh against the child nodes depends on the bounding volume we are using to represent our meshes. If we are representing the bounds of our meshes with a bounding sphere for example, then determining whether a mesh intersects the bounding volume of a child node becomes a simple Sphere/AABB test. If the mesh's bounding volume is represented as an AABB (as ours will be), then the mesh/node test is an AABB/AABB test. As you might image, this is quite a bit faster than performing a Polygon/AABB test for each node and for every polygon in the mesh.

As an example, we might have a level constructed from 100 meshes, each containing 1000 polygons. If we were to compile our oct-tree or quad-tree at the polygon level, compiling the tree would mean traversing the tree with 100,000 polygons until they all eventually ended up being stored in their respective leaf nodes (perhaps with lots of clipping being performed). This is the approach we used in the quad-tree example earlier and as you can imagine, an awful lot of Polygon/AABB tests have to be performed at each node before the tree is fully compiled. If we decided instead to simply build the tree as a mesh tree, then all we would need to do to build the tree is send in the 100 bounding volumes of our meshes. It is the bounding volumes that would be classified against the nodes of the tree and would eventually end up in leaf nodes. In a tree used purely for the partitioning of mesh data, the leaf structure could contain a linked list or array of all the meshes stored in it.

> **Note:** It is common for a mesh that is large or just situated very close to the split planes to be assigned to multiple leaf nodes. There are strategies that exist to minimize this problem and guarantee that a mesh fits completely in a single node. Thatcher Ulrich's discussion of 'loose' oct-trees in *Game Programming Gems 1* (2001) would be a worthwhile read if you are interested in exploring this further.

A mesh tree is ideal for entities that are constantly having their position updated in the scene. If a mesh moves in our scene, we can simply unhook it from the leaf nodes to which it is currently assigned and feed it in at the top of the tree, and traverse the tree again with its bounding volume until we find the new leaf nodes that its bounding volume intersects. A mesh oct-tree for example might be ideal for a space combat game for example, where space could be uniformly subdivided into cubes (leaf nodes) and as a space ship mesh moves around the game world, the tree is traversed again to find the new leaf nodes it is contained within. A mesh is only rendered if one of the leaves it is contained in exists inside the frustum; otherwise it (along with all its polygons) is rejected from the rendering pipe.

When performing intersection queries on the tree (e.g., testing our swept sphere against the tree) the polygons of a mesh only have to be individually tested for intersection if the intersection volume intersects the bounding volume one of the leaf nodes in which the mesh is assigned. Since an entire mesh might exist in many leaf nodes simultaneously, it is important that the system establish some logic to avoid rendering the mesh more than once or checking its polygons for intersection more than once during a single query. As mentioned, this can also be a problem with a polygon tree when the polygons have not been clipped to the nodes. However, this is a relatively smaller problem with the polygon based tree since it would typically be only a handful of polygons here and there that would be rendered or queried multiple times. This might be something we are prepared to accept since it would probably not impact performance by a significant amount in the typical case. For a mesh tree however, it is crucial that this problem be resolved. The bounding volume for an entire mesh may well span dozens of leaf nodes, and if all those leaf nodes were visible, we certainly would not want to render the entire mesh with its thousands of polygons multiple times. The same is true for polygon queries. If our swept sphere intersected dozens of leaf nodes which all contained the same 20,000 polygon mesh, we certainly would not want to perform the swept sphere/polygon intersection tests for 20,000 polygons more than once, let alone dozens of times. Suffice to say our game would become less than interactive at that point.

14.6.3 Mesh Trees vs. Polygon Trees

It is easy to be seduced by the ease with which a mesh tree can be constructed and the speed at which the tree can be dynamically updated when objects in the scene move. To be sure, in many cases it will absolutely be the right choice for the job, whether you are creating a quad-tree, oct-tree or kD-tree (or any other type of tree – like a sphere tree, which is also very popular for dynamic objects). Of course, this decision will also be based on the format of your input data. If the scene is built from a set of mesh objects then the mesh tree would be much easier to implement. If the scene is represented as a static world space polygon soup (a little like the static geometry imported from an IWF file) a polygon tree is probably the best bet.

We do have to be aware of the disadvantages of a mesh tree however, since it may not always be a better choice than a polygon tree if the scene is comprised of multiple static meshes. Ease of construction and update speed does not always come without a cost. Whether that cost is significant depends of the specifics of your application. When you have dynamic objects in your scene, they absolutely have to be connected to the tree at the mesh level, so that they can have their positions within the tree updated in an efficient manner. When you have a scene represented as a static polygon soup, that will most likely fit in well with the polygon level subdivision techniques we have discussed. However, when the scene is comprised of multiple *static* meshes which do not need to have their positions updated, should a mesh tree or a polygon tree strategy be used?

To understand the disadvantages of a mesh tree, let us imagine a situation where a single large mesh consisting of 20,000 polygons spans dozens of leaf nodes (perhaps a terrain mesh). To be sure, this is a worst case example, but it will help to highlight the potential disadvantages of the mesh tree at performing frustum culling and polygon queries. Now, let us also imagine that we have a ray that spans only two leaf nodes which we wish to use to query the tree. In other words, when the ray is sent down the tree to retrieve the closest intersecting polygon, ideally only the polygons that exist in those two leaf

nodes would need to be tested. With the mesh tree, the leaf nodes might contain indices or pointers to the mesh objects that are attached to that leaf. When a leaf is found to be relevant to a query, every polygon in every mesh attached to that leaf will need to be queried at the polygon level. In the case of our large example mesh, only a very small subset of polygons (say 100) actually reside within those two leaf nodes, but we have no way of knowing that with a mesh tree. We simply know that some of the mesh's polygons may intersect the ray. Therefore, we would have to perform per-polygon intersection tests on the entire mesh (all 20,000 polygons) even though only a handful are within the space represented by the leaf nodes in which our ray currently resides. That certainly is not good for performance, and if this mesh was not dynamic, it would probably be wise for a broad phase collision implementation to partition this mesh at the per-polygon level.

> **Note:** The mesh itself could also be internally managed using some form of partitioning structure, even though it exists in a higher level scene mesh tree as single object.

In the case of a polygon tree where we have not performed clipping of the polygons, things are certainly a whole lot better as only a very small subset of polygons of the original mesh will be assigned to those two leaf nodes. This might not be an exact set of polygon data that fits inside the leaf nodes because some polygons may only be partially inside, but as long as we test those polygons and make sure that we do not test a single polygon multiple times if it exists in multiple leaf nodes, we only have to query the polygons that are (to some extent) inside the bounding volumes represented by the leaf nodes.

In the case of a polygon tree where clipping has been performed, we only ever query or render polygons that are completely contained within the leaf nodes that are intersected by the ray. The following diagram illustrates the disadvantages of using a mesh tree in such a situation where a ray/polygon query is being performed on the tree and the mesh is large enough to span multiple leaf nodes.

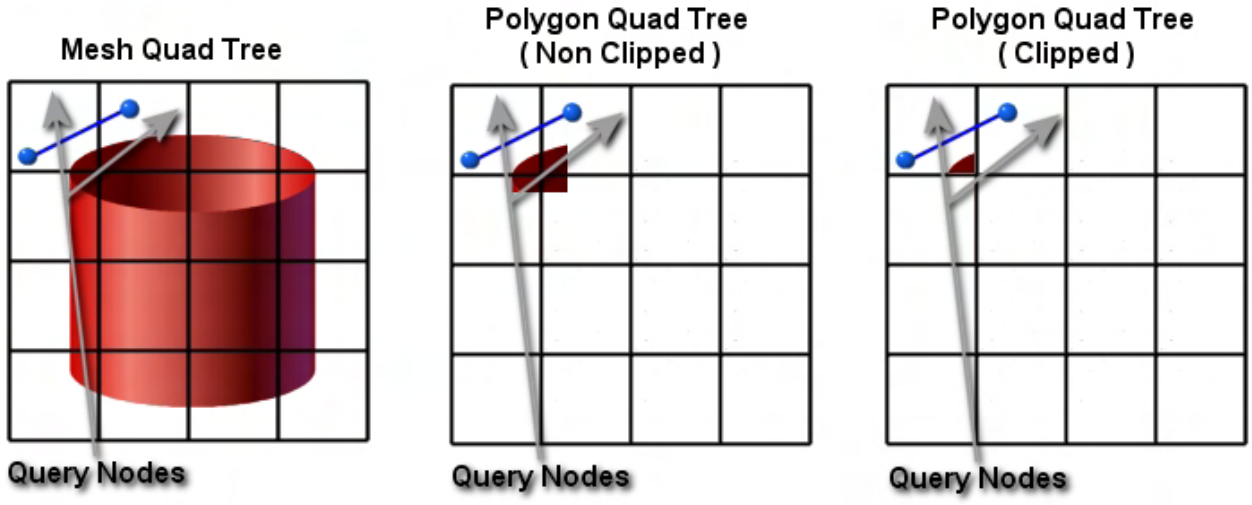


Figure 14.38

In this example we show a quad-tree as it is easier to represent on a 2D sheet of paper, but the same logic holds true for all trees. On the left we can see a single mesh of a cylinder consisting of our 20,000 polygons. As you can see, it spans a large region of the overall scene and therefore, is assigned to multiple leaf nodes. In this particular example the mesh is partially inside every single leaf node so would be assigned to each. The two blue connected spheres represent the ray being used to query the

scene and we can see in all three examples that it only ever intersects two leaf nodes in the top left corner of the scene. Ideally, only the polygons of the mesh that exist in those two leaf nodes should be queried. In the case of the mesh tree, we would have to test all the polygons of the cylinder for intersection, even though we can see that only a very small number of the mesh's polygons exist inside the two queried leaf nodes. If we were to render this mesh tree, the entire mesh would be transformed and rendered by the pipeline every single frame because the cylinder exists in every leaf node. This obviously means that this mesh would never be frustum culled and would always be rendered in full.

The middle image shows a non-clipped polygon quad-tree. Because the leaf nodes contain only the polygon data that are either inside or partially inside those nodes, only a very small portion of the mesh would need to be queried for collision detection with our ray. As you can see in this example, although the polygon data does not always exactly fit the leaf nodes (notice the overspill in the diagram) this does not affect the efficiency of our collision detection routines because only polygons that have some of their area inside the leaf nodes are tested. This would obviously be a lot quicker than querying every polygon in the entire mesh as was the case with the mesh tree. We can also see that during the frustum culling pass through the tree, most of the mesh's polygons will be rejected and only a very small subset would be rendered if the camera was positioned such that only a handful of leaf nodes existed inside the frustum. Some of those polygons may lie partially outside the frustum and would need to be clipped by the pipeline, but this would be negligible to performance in the typical case.

In the rightmost image we see the clipped version of the quad-tree where the leaf nodes contain only polygons that fit exactly inside them. If any polygon was found to be spanning multiple leaf nodes during tree compilation, the polygon would be clipped to those nodes and the fragments assigned to their relevant leaf nodes. This provides very little benefit during the polygon query phase as the same number of polygons would still need to be queried. The fact that those polygons have been clipped and are smaller has no affect on the speed of the ray/poly intersection routines. That is assuming of course that in the case of the non-clipped tree, provisions are made so that a polygon that spans multiple leaf nodes is not queried multiple times. We will use an application timer to assure that this is the case, which we will explain later in this lesson.

It is true when looking at the clipped polygon tree that less of the scene would need to be rendered because only polygon data that exactly fits inside the leaf nodes would be rendered if that leaf node is visible. The same number of polygons would still need to be rendered in the typical case and more polygons in the worst case where everything is visible (because of the splitting of polygons). We can imagine that if a triangle spanned a leaf node during the tree building process, that polygon could be clipped into two fragments, a triangle and a quad. We have now created three triangles where one previously existed. Frankly, on today's hardware, we would probably see little to no benefit from this type of tree versus the non-clipped case. In some cases, we may even see a *decrease* in performance due to the larger polygon count and increased number of DrawPrimitive calls. A clipped tree still provides benefits in other areas where we absolutely must know which section of a polygon lay inside a bounding volume, and you will see later that this is definitely the case when we create a node based polygon aligned BSP tree. For example, a non-clipped tree means that a single polygon may be assigned to multiple leaves. Thus, during a rendering pass we may render it multiple times. Sure, we can embed some logic that prevents a polygon that has already been rendered from being rendered again during a single traversal, but then this would introduce a per-polygon test. We want to render polygons in huge batches as quickly as possible and certainly do not want to be doing a per-polygon test to determine if

we should render each one. On modern hardware, it would generally be quicker just to render those polygons again. However, what about collision queries?

If we are using a non-clipped polygon tree for our broad phase collision system then we still have this problem to overcome. We certainly do not want to test the same polygon in the expensive swept sphere/polygon test more than once. This involves the costly transformation of the polygon into eSpace and the various intersection routines to determine if an intersection occurs. As discussed, we will implement a system so that the collision system can query non-clipped polygon trees without querying a single polygon multiple times.

So we have seen that while the mesh tree has some advantages, it certainly has disadvantages over its polygon based counterpart in other areas. The examples given above are somewhat extreme because if your meshes are of a size such that they fit inside a leaf node in their entirety, this situation will not arise. This situation only becomes problematic on a dramatic scale when the meshes in your scene can span many leaf nodes and have high polygon counts.

Object/Object Collision Testing

Spatial hierarchies can also speed up the collision detection between multiple dynamic objects in our scene. Let us imagine that we have created a space combat game and the region of space that our meshes will occupy has been spatially divided into uniform sized leaf nodes using an oct-tree. Our scene might very well have many spaceships all flying around in space, and while the polygon queries we have examined in this chapter show how a spatial hierarchy can be used to determine which polygons are intersected by rays or bounding volumes or swept spheres, what about the fact that every dynamic object in our scene might collide with every other dynamic object in our scene? In other words, collision detection does not have be performed only between our player and the static scene and any dynamic objects it may contain, but collisions can also be performed to make sure that the dynamic objects in our scene to not collide with one another.

It is quite common for dynamic objects to use simplified bounding volume collision detection in a game. In the interests of speed, when two objects collide in our world, we are very rarely interested in which polygons from each mesh intersected one another. Usually we will perform collision tests between meshes using bounding volumes such as spheres, OBBs or AABBs. Testing collision detection between two bounding volumes is generally much cheaper than testing two objects at the polygon level, which makes it ideal for a broad phase collision step for dynamic objects. If the bounding volumes of two objects do intersect, then you have two choices: you can either treat this as a collision and allow the meshes to respond to the collision accordingly, or you can use the intersection result as a test to see whether the meshes should be queried at the polygon level to make sure that an actual polygon/polygon intersection occurred between the two objects. You will usually find that using the bounding volume intersection result as a test for inter-object collision will suffice. Even if the objects did not physically collide (i.e., their polygons did not physically touch), everything is usually happening so quickly in a 3D game that the player will rarely notice the difference. Of course, the success of this approach depends on the bounding volume being used and how tightly it fits the actual object.

The quickest bounding volumes to test for intersection are spheres, but these are usually the bounding volumes that least accurately fit the shape of the object. When using a bounding sphere around a non-spherical object or an object that is much longer along one of its dimensions, the player may see two meshes respond to a collision of their spheres, even though it was quite visible to the player that the objects did not actually collide. If you are simply using the result of the bounding volume test to progress to the more accurate per-polygon testing process, then this is not so much of an issue. In this case, the sphere tests act more like a broad phase for objects that cannot possibly collide and therefore do not need to be tested at the polygon level.

A generally better fitting volume with fast intersection testing is the AABB. Therefore, one approach is to use an AABB for each of your objects and calculate it each time the player's position/orientation changes. While this might sound extremely slow this does not involve having to compile the AABB from scratch by testing each of the mesh's vertices whenever it rotates. This step only has to be performed when the AABB is first constructed and from that point on we can use some very quick and efficient math to recalculate a new AABB for the object in its new orientation. We will discuss such a technique later in the lesson but for now, let us assume that we are using AABBs for our dynamic objects for the remainder of this discussion.

One of the best fit bounding volumes is the Oriented Bounding Box (OBB). It is similar to an AABB in that it is also a box, but it differs in that it is calculated to try to fit the general shape of the object. Rather than be restricted to using the primary world axes, OBBs use a system closer to object local space to calculate the size and orientation of the box based on the general spatial distribution of vertices in the model. OBBs are a very popular choice for collision systems, especially ones that only wish to respond to collisions between bounding volumes and not perform actual polygon collision detection (although it can obviously be done if desired). Testing collisions between two OBBs is a fair bit slower than intersection testing two spheres or AABBs, but the results are going to be more accurate since the bounding volume is a better fit. They can certainly be worth it when the situation calls for it. Indeed an OBB (or multiple OBBs arranged hierarchically) can be constructed to bound a mesh so tightly that it allows us to do away with Polygon/Polygon' testing altogether in many cases and therefore greatly speeds up our ability to do more realistic collisions. Of course, there will still be times when the objects will react to a collision that did not actually happen between their physical geometry, even though their bounding boxes collided. However, this will likely go unnoticed by the player if the OBBs fit nicely. OBB construction and OBB intersection testing is fairly complicated from a mathematical perspective and takes more cycles to perform than simple AABB/AABB intersections. For the moment, we will hold off on discussing OBBs and revisit them a little later in the training series.

Now, let us imagine that we have decided that our space game is going to use a mesh tree and will use AABB/AABB intersection testing for the collision detection of its dynamic objects. Let us also imagine that in our space combat game a mighty battle is taking place between 500 ships. One approach to dealing with their collision status might say that for every frame, we loop through all 500 meshes and perform an AABB/AABB intersection test with the other 499 meshes in the scene. This seems to be a rather non-optimal approach since typically only a handful at most will actually be colliding. In fact, it is likely that most of these meshes will not even be in the same leaf nodes as any other ships and could not possibly collide at all.

This is where the hierarchy system can begin to show its true colors. The leaf structures in a mesh tree could contain a list of meshes that are inside it (or partially inside it) and depending on how much we subdivide our scene and the size of our final leaf nodes, there is a good chance that at most, a single leaf will contain only a few meshes. There is also a good chance that many leaves may contain only one mesh. Therefore, instead of naively testing every dynamic object against every other dynamic object, we could instead just test it against the mesh lists for the leaves the source mesh is contained within. For example, if the dynamic object structure could retrieve a list of leaves it is currently contained within, then we know the object could not possibly collide with anything that is not in one of those leaves. Therefore, we just have to access each of those leaves and test against the meshes stored in the mesh lists of those leaves.

Although we will be implementing our mesh tree strategy in a different way, the following code snippet demonstrates this concept. pDynamicObject is assumed to be an object that we wish to test for collision against other dynamic objects that exist in the same leaves. It is assumed to contain an array (LeafArray) of all leaf indices it is currently assigned to. Obviously, if any of these leaves contain only one mesh then this must be the only mesh in that leaf so no collision can occur there and we move on to process the next leaf it is contained within. Otherwise, we loop through and test the mesh's volume against every other mesh in the leaf using the ProcessCollision function. We might imagine how such a function would compare the AABBs of both meshes and perform some response if both objects collide. This is a very simple example using pseudo structures and functions and exists to give you a basic understanding of the way the information stored in a spatial hierarchy (especially a spatial tree that has the ability to also link dynamic objects to leaves) can be used to accelerate a multitude of processes.

```
for ( int i = 0 ; i < pDynamicObject->LeafCount; i++ )
{
  if ( pDynamicObject->LeafArray[i].DynamicObjectCount > 1 )
  {
        pLeaf = &DynamicOject->LeafArray[i];

        for ( int i = 0; i < pLeaf->DynamicObjectCount; i++ )
        {
            CDynamicObject *pTestObject = pLeaf->DynamicObjects[i];

            if (pTestObject!=pDynamicObject) ProcessCollision( pDynamicObject,
                                                               pTestObject );

        } // End for each object in current leaf

   } // If more than one object in leaf

} End for each leaf the dynamic object is currently in
```

Using the system described above, even if you had 10,000 meshes in your scene, if none of the meshes shared a single leaf node, no bounding volume collision tests need to be done. That is obviously much quicker than just blindly testing every object with every other object for collision. How much quicker? Well in the 10,000 mesh case you would have to do 49,995,000 tests! This is because you are calculating an addition series that amounts to the following formula:

$$\#tests = \frac{n(n-1)}{2}$$

If there are *n* objects, you do not test an object against itself and you only test between objects once (i.e., after testing A against B, there is no need to test B against A since the result is the same), thus cutting the number of tests in half. Even still, with a modest 50 objects in your scene, that is 1,225 tests that will need to be run. Granted, you will also have to weigh the cost of object insertion into the hierarchy versus the linear approach to determine which is more efficient for your system. However, if you are running the objects through the tree anyway (e.g., for their static collision testing) then you would not have to worry about adding overhead for traversals since that cost has already been incurred. In that case, the hierarchy will win out since you will only test objects that share leaves.

14.6.4 Combining Polygon Trees and Mesh Trees

So with the differences between the various tree types distinguished, which are we going to use in our lab project? Are we going to use a mesh tree, a clipped polygon tree, or a non-clipped polygon tree? Actually, we are going to allow our application to support all three tree types. We will have a single tree (such as a quad-tree for example) that compiles any static polygon/mesh data such that it is stored in the leaf nodes at the polygon level (polygon tree), but will also keep track of which leaves contain our dynamic objects (mesh tree). The basic building strategy will be as follows…

First we will load the data in from an IWF file. Any static data that is loaded from the IWF file (GILES™ internal mesh geometry stored in world space) will be added polygon by polygon to the spatial tree we are currently using. Once we have added all the polygons to the tree, the tree object will have a big list of all the static polygons we wish to have compiled into a spatial tree. The application will then call the tree's Build function which will instruct the tree to partition the space described by its polygon list. At the end of the build process, the space containing the polygon data will have been subdivided into a number of leaf nodes and each leaf will contain an array of the polygon pointers contained inside it. Further, we will allow the application to set a flag that instructs whether this polygon tree should be compiled using clipping. If not, then during the build process, a polygon will always be assigned to any leaf nodes it spans. If clipping is enabled, a polygon will be split if it spans leaf boundaries so that each leaf will only contain pointers to polygons that exactly fit within their volume.

It sounds like we have basically just decided to build a polygon tree, but that is not the full story. The leaf nodes of the tree itself will also be allowed to store pointers to dynamic objects (in actuality, a more generic structure will be used so that literally anything can be stored, but the end result will be the same). We can simply send the bounding volume of the dynamic object down the tree and store its pointer in any leaf nodes it is found to be contained within. We can also add some functionality to retrieve a list of these objects from the leaves as needed for various purposes (collision detection, rendering, etc.).

So we can see that our tree is basically both a mesh tree and a polygon tree although the tree will only directly store the static polygon data (meshes and other objects will simply be stored as pointers). In our

next chapter we will learn that the tree will even know how to render the polygon data. That is, in our main scene render function we can just call the ISpatialTree::DrawSubset function and it will only render the polygons contained in the leaves which are visible and have polygon data belonging to that subset. Since the objects can easily find out which leaves they are contained within, determining their visibility prior to rendering will also be quite straightforward.

Although this might all sound a little complicated you will see when we examine the code that it really is not that bad. In prior applications we had a single rendering pool, an array of CObjects which had to be rendered. Now, we essentially have two pools. We have a tree to render (which will contain all the static polygon data) and the array of CObjects (although they are ultimately linked into the tree as well, in a manner of speaking). However, only dynamic objects will be stored in the scene's CObject array from now on. All static geometry will be contained inside the tree and rendered/queried using the tree's rendering/querying methods. The tree will know how to render its polygon data efficiently and in a hardware friendly manner.

The tree will also expose methods such as ISpatialTree::CollectLeavesAABB. This method when passed an axis aligned bounding box will return a list of leaves intersected by that AABB. Functions like this can be called to run queries on any number of external objects to query their position within the tree even if the tree is not aware of their existence. Dynamic objects and terrain blocks will all use this function to determine whether or not the regions of space they are contained within are currently visible without their geometry ever having to be compiled into the tree. We could build a pure mesh tree (a space scene for example) by compiling an empty tree (subdividing empty space into a fixed number leaves) and then associate dynamic objects and terrain blocks with their leaves using this function.

> **Note:** The lab project that ships with this chapter imports its data from an IWF. The internal mesh data is treated as static polygon data and is compiled directly into the tree. Any external X file references will (as always) be treated as dynamic objects, loaded into CActors and stored in the scene's CObject array as usual. However, they will still be tracked by the tree itself. If you have a static X file that you would like to have compiled into a tree at the polygon level, then simply import it into GILES™ and export it as an IWF file. The X file data will become static geometry in the IWF file. If you have dynamic X file objects that you would like to place in the scene then place them in GILES™ as reference entities. Our application loading code will then know not to compile them into the tree at the polygon level and they will be treated as dynamic objects.

14.7 Areas of Interest

Before we start to examine the code there is one more matter that we must discuss and make provisions for if we are to make our tree classes as useful as possible. We must allow for the spatial tree to be instructed to partition space where perhaps no geometry exists during the compilation process.

Up until now we have discussed that during the recursive building process we will decide to stop subdividing nodes down a given branch of the tree if that node's bounding volume is very small or if only a small number of polygons exist there. These settings are all going to be configurable, so you may decide that you wish to have leaves with the capacity to store 1000s of polygons or perhaps only a few dozen. The same is true with the minimum node size setting. You may decide that you would only like

to only modestly subdivide your scene and set the minimum leaf size (the size at which a node is no longer further partitioned) to be a rather large volume of space. Alternatively you may instead decide to make the tree partition space is to lots of very small leaves. The choice is yours and trial and error will often be as good as any deductive reasoning when deciding for a given level what you strategy should be. However, there is still a problem with this procedure which, if nothing was done to remedy it, would make our tree classes useless as mesh only trees.

The problem is that the bounding boxes of the nodes (including the root node) are computed during the build process by finding the tightest bounding box that fits around the polygon data living in that node. In the case of the root node, this means that the bounding box will be calculated using the static polygon data that will be stored in the tree itself. We do not bother factoring the positions of any dynamic objects at this point for a few reasons. First, building the tree based on the positions of objects that will change in the very next frame makes little sense. Second, it is quite common in a game level for dynamic objects to be spawned on the fly in response to some game event, which means the dynamic objects would not even be available during the building of the tree. Finally, we have also stated that with our design, the tree will have no knowledge of what polygons a dynamic object contains. We will provide a means for determining which leaves an object is in, but those objects will manage their own polygon information.

Of course, we do not always want our spatial tree to be as small as the static polygon data that will be fed in. This is definitely the case if we wanted to build an empty tree (i.e., partition empty space) and then use it to manage the collision querying and rendering of dynamic objects. The root node would have been passed no static polygon data, its bounding box would have zero volume, and it would be the only node in the tree because it has no polygon data and is infinitesimally small.

Consider once again the creation of a space combat simulation where ships will be flying around and attacking each other. Imagine that there are no planetary bodies, so all we have is an area of space that will, at some point, contain many dynamic objects. The obvious thing to do here would be to first partition the empty region of space into an oct-tree, for example. Essentially, we wish to create a tree that will subdivide the space that the ships will be permitted to fly around in, but at build time we wish this tree to be empty. That is, none of the leaf nodes will contain any static polygon data. Why? Because in this example there is no static polygon data; we simply want to partition space for the benefit of our dynamic objects. Once the tree was built, each time a dynamic objects position is updated, we can feed its AABB into the tree (using the ISpatialTree::CollectLeavesAABB method). This method determines which leaf nodes contain the dynamic object and store this information internally (as well as export it to the caller).

Figure 14.39 shows how three dynamic objects might look when placed within the region of space subdivided by the oct-tree. Remember, the oct-tree itself would still think it was an empty tree as it has no polygon data stored at its leaf nodes, but it does update itself each frame and determines which leaves are currently inside the frustum and are therefore visible. Once the dynamic objects are introduced, it can keep track of this information internally using some generic data structures and return the list of visible leaves as needed, or even return the list of visible dynamic objects if desired. Incidentally, as Figure 14.39 also shows, there would be no need to test the dynamic objects against each other for collision as none of them exist in the same leaves.

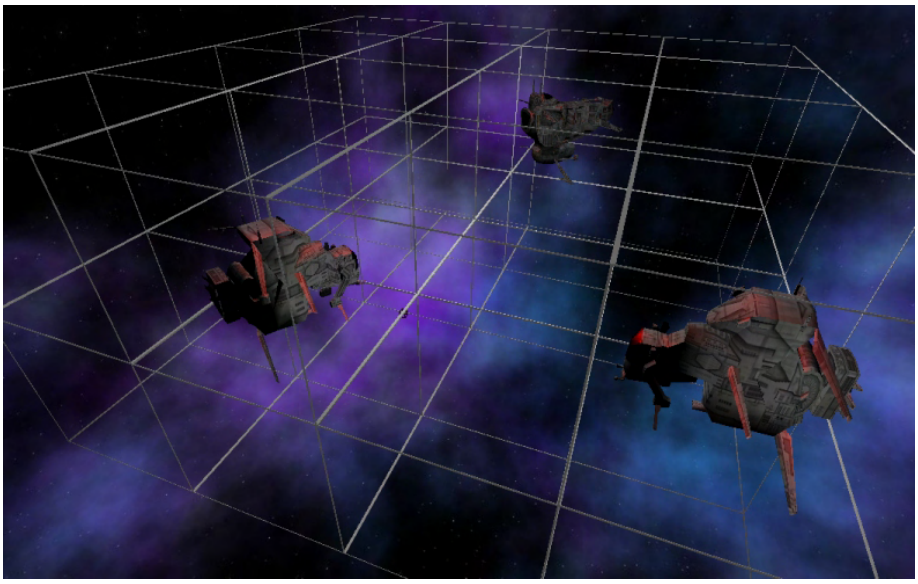


Figure 14.39

So we can see that we need the ability to create empty trees of a specific spatial size specifically so that we can control the region of space that will become part of the tree, even if that region is not described (or fully described) by the static polygon data from which the tree is generally built. Even if there is static data with which to build the tree, we might not want to simply compile only this exact region into the tree. We still may want the sky above a cityscape model to be partitioned and stored in the tree so that flying objects can enter the cityscape's airspace and benefit from the tree as well.

In the example shown in Figure 14.39 we might pass a large AABB into the tree building function which specifies a region of space that must become part of the tree and be spatially subdivided. We refer to these in our system as *detail areas* (or *areas of interest*). The tree can then continue to carve this space just as if there was polygon data contained in it. Only in this instance, the leaf nodes would contain no polygon data.

Another example where this is useful is when using a scene that contains a CTerrain object. As we know, a CTerrain object is constructed from a series of CTerrainBlocks where each block is a separate mesh. As terrain blocks could be quite efficiently frustum culled by testing their bounding boxes against the frustum it would be wasteful to store copies of the every triangle in the terrain in a tree. We saw in the previous lesson how our collision system essentially avoided doing the same thing by converting the swept sphere's AABB into terrain space and building only the relevant triangle data on the fly that needed to be tested. The same should also be true for our rendering; it would be overkill to store the triangle data for each terrain block in the tree when we could instead treat each terrain block as a dynamic/detail object. That is, we could pass the AABB of the of the terrain blocks down the tree and have pointers to them stored in the leaf nodes in which they belong. Only when those leaf nodes are visible does the terrain block render itself.

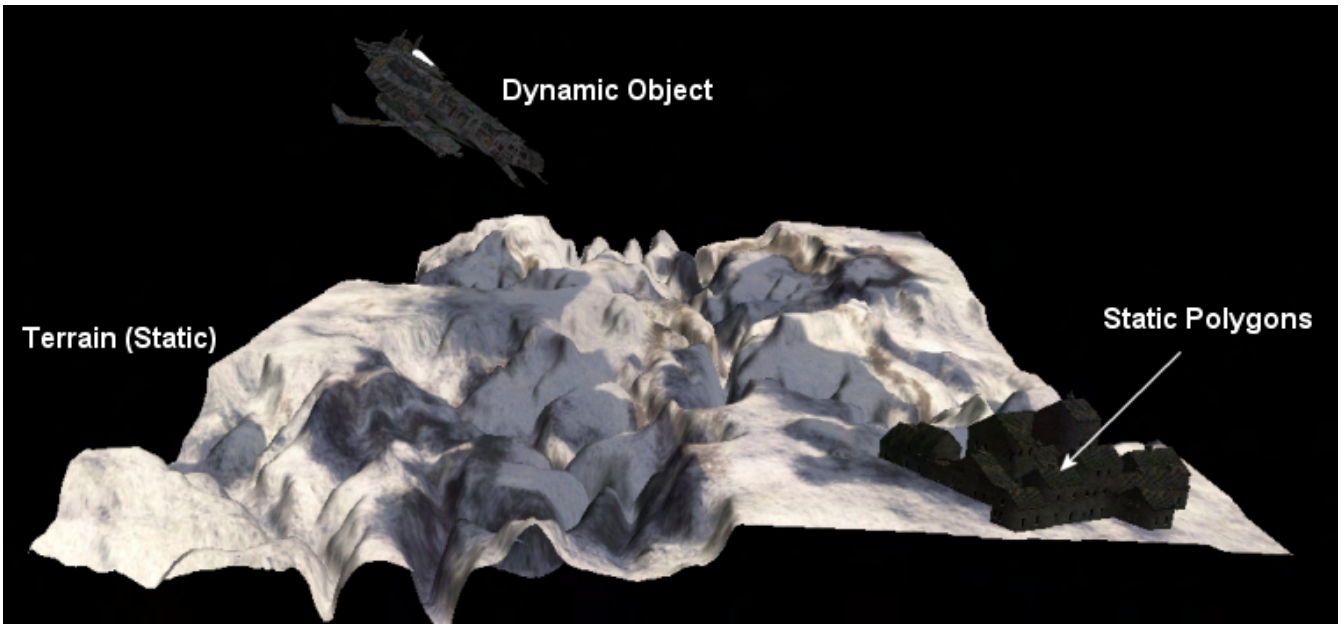


Figure 14.40

This is a good example of why we need the ability to add areas of interest (detail areas) to the tree. In Figure 14.40 we see a scene that contains a terrain (composed of multiple terrain blocks), a dynamic object in the sky above the terrain, and a small settlement made using static geometry. Because we do not wish to assign the terrain triangle data to the tree and we do not wish to factor in any temporary position of the detail object during the tree building process, the only polygons initially input to the tree building phase would be the static polygons labelled on the image. However, using the building strategy we have discussed thus far, the root node of the tree would have its bounding volume calculated only to be large enough to contain those static polygons. In Figure 14.41 we show that this would create an oct-tree that actually subdivides only a small section of the overall space in the scene we wish to use.

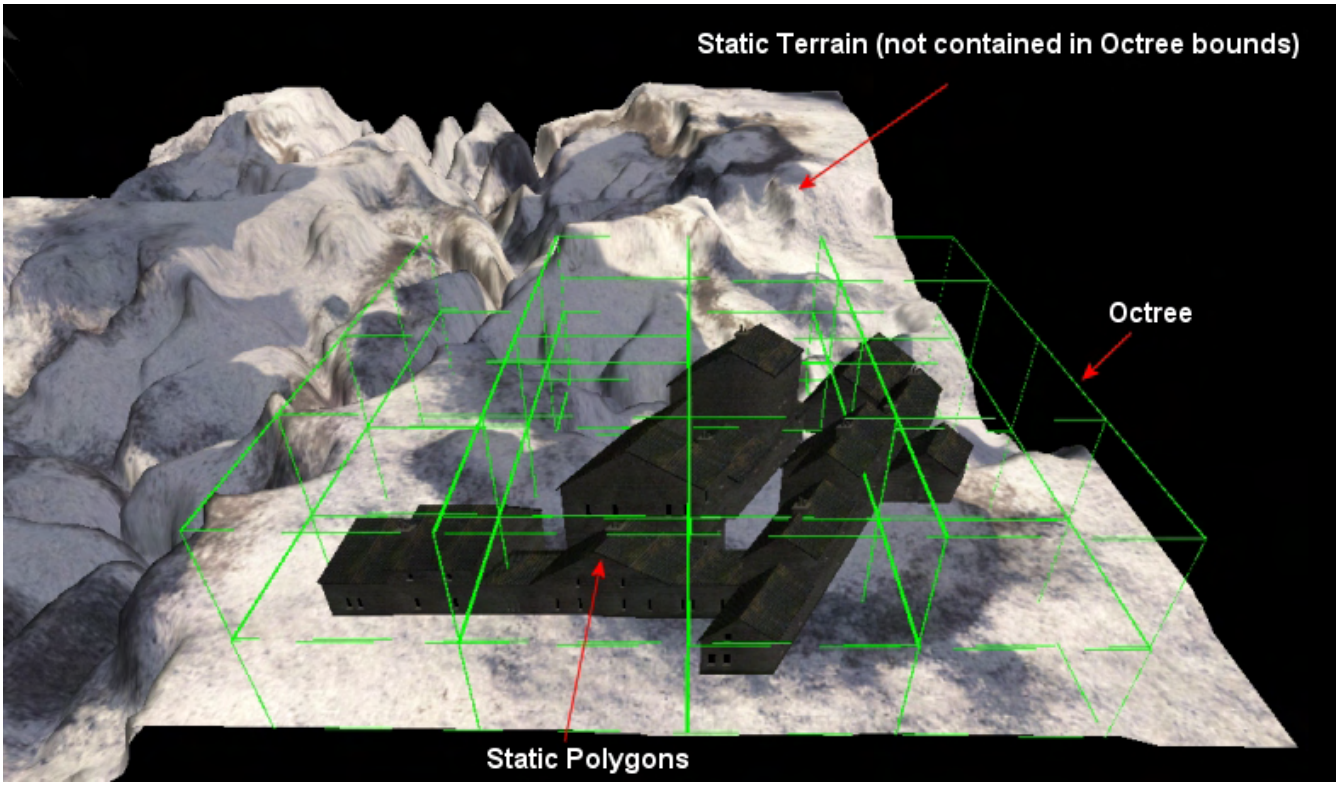


Figure 14.41

Now imagine that after the tree has been built, we decide to send the bounding volumes of each terrain block down the tree to find out in which leaf nodes they belong. We have a problem -- most of the terrain blocks are not even within the bounds of the root node's volume, so they would be found to be outside the tree and would not be able to benefit from the tree's various properties. The same is true once we place our spaceship (dynamic object) in the skies above the terrain. The detail object would have its AABB passed down the tree each time it is updated and we would find that it is contained in no leaf nodes since it is not within the area of space occupied by our partitioning scheme. This might mean the object does not get rendered at all (because it is technically not inside a visible leaf) or that such objects must be individually tested and rendered (depending on the rendering strategy you are using).

Alternatively, using the same detail area example seen above, we could pass into our tree building function a large bounding box that encompasses not only the stone settlement but also the terrain and all the sky above the terrain we intend to use for our dynamic objects. This would force the root node of the tree to be of a size that is large enough to contain the entire scene and each level in the tree would further subdivide this space.

If we imagine that we have done just that and that our tree building strategy is updated to calculate the correct size of a node's bounding box (not only on the static polygon data in its list, but also any detail area AABBs that have be registered with the tree object prior to the commencement of the build process), we can see the results would be exactly as needed as shown in Figure 14.42.

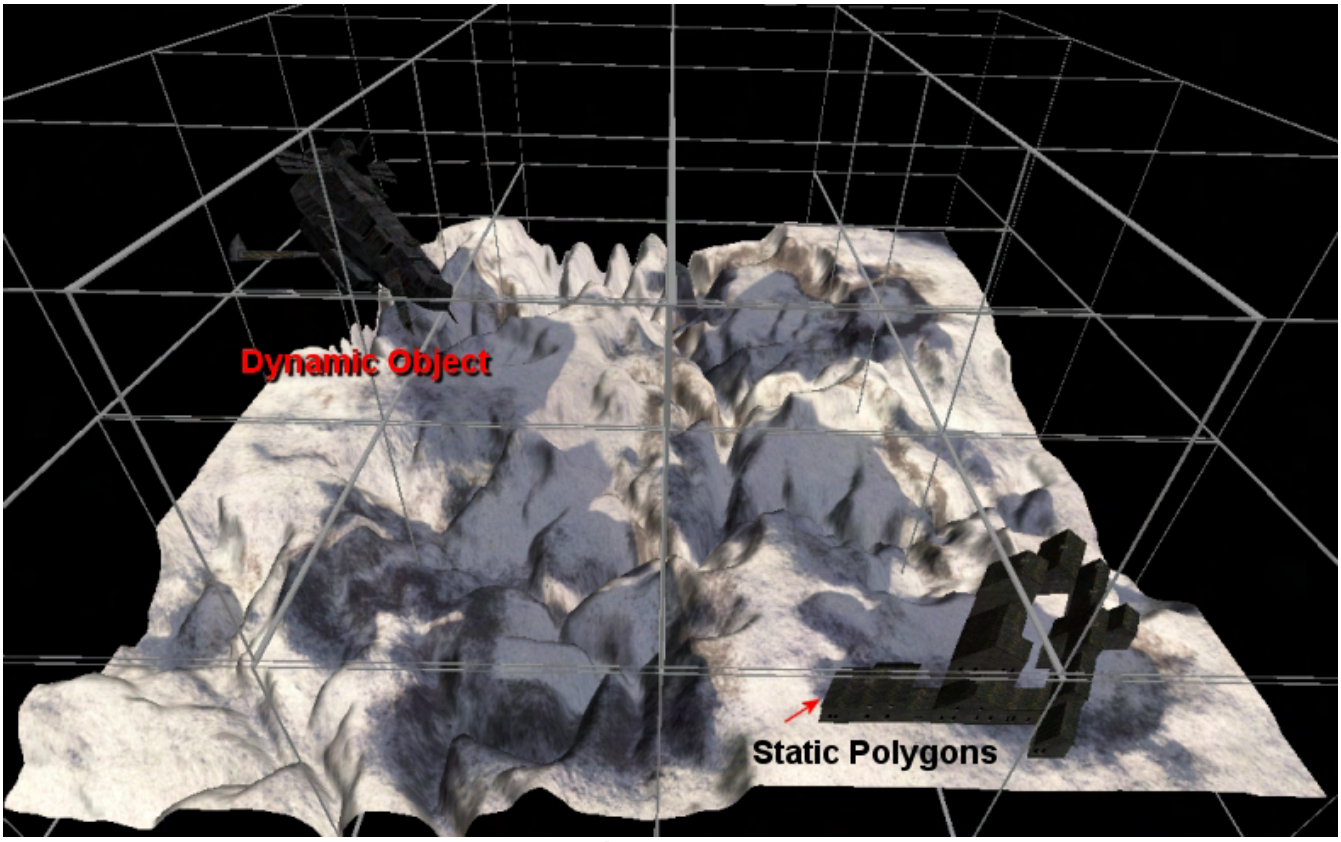


Figure 14.42

This really does solve all of our problems and allows us to build trees that either completely partition empty space or that partition space that is not necessarily fully occupied with static input polygon data.

Of course, this does not mean that we are restricted to passing only a single AABB as an area of interest. We might pass in multiple areas of interest at different positions in the level identifying areas where nodes should be partitioned even if no polygon data currently exists there. During the building process the condition used to determine whether a given node should be made a leaf node is now either of the following:

1. The bounding volume of the node is sufficiently small.
2. The bounding volume contains a small amount of polygon data *and no area of detail*.

The second condition (polygon count) has subtly changed, but it is an important one. Before we said that we would make a node a leaf if the number of polygons in that node's list is below a certain threshold amount. But now, even if there are no polygons in the node, but there is a detail area intersecting that node and the node has not been subdivided down to our minimum leaf size, we will continue to subdivide. This means that not only do these detail areas allow us to include space in the tree that is outside the region described by the static polygon set, but they also allow us to specify areas at any point in the level where we would want the space to be more finely partitioned down to the minimum leaf size.

It will be easy to add these areas of interest to our tree system. Our tree classes will expose a function called AddDetailArea which allows the application to register any bounding volumes with the tree as areas of interest. These areas must be registered prior to the Build function being called so that the construction of the tree can them in to its computations when determining the bounding boxes of each node and figuring out whether or not a node should be further subdivided. Further, we will also provide the ability to register a context pointer with the detail area which can point to application specific data. This could, for example, point to a terrain block that needs to be rendered or a sound that should be played whenever the player is in that leaf (or if that leaf is visible). Although we will actually have no need to use this context pointer in our first implementation, it is handy to have around and is something we will undoubtedly find useful moving forward.

14.8 Tree Balance

Before we begin to code, we must be aware of the implications of creating a wildly unbalanced tree. A perfectly balanced tree is a tree where all leaf nodes exist at exactly the same level in the hierarchy. Although it is rarely possible to achieve a perfectly balanced tree (while not subdividing empty space and introducing unnecessary nodes) it is something we wish to strive for as much as possible in many cases because it allows us to keep tree traversal times consistent during queries. The last thing we want is one part of our tree to be many levels deep and another section of the tree to be only a few levels deep. This will cause inconsistent frame rates when querying the deepest parts of the tree. Figure 14.43 shows an unbalanced quad-tree.

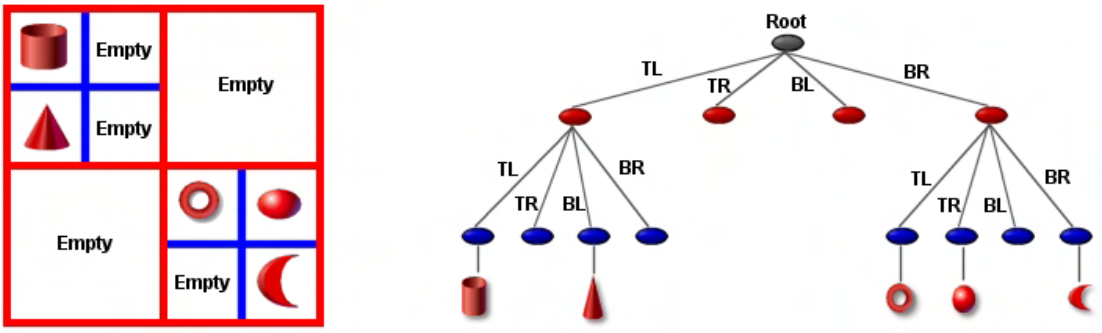


Figure 14.43 : Unbalanced Quad-tree

In this example we can see that only two of the four children of the root node have geometry to subdivide and as such, nodes TR and BL (which have no geometry passed to them) become empty leaf nodes (terminal nodes). However, the polygon data passed into the TL and BR children of the root do have geometry that falls within their bounds, so their volumes are further divided. Although this is a simple and small example, we can see that queries would take longer to perform in the top left and bottom right quadrants of the scene than in the top right and bottom left quadrants. That is, we would reach the terminal nodes quicker for the top right and bottom left quadrants. The difference in query times in the real world is much greater when we deal with trees that are many levels deep. It is better to have a scene that can be rendered/queried at a consistent 60 frames per second than it is to have a scene where the frame rate runs at 150 frames per second in some areas and at 10 frames per second in others.

This is a somewhat drastic example but the point is that the application will benefit from smoother physics, collision response, and general movement if the frame rate can stay relatively consistent throughout.

If we imagine that we had the ability in the quad-tree to slide the clip planes to arbitrary positions at each level, we might imagine that we could end up reducing this to a two level tree where all four children of the root (2^{nd} level) has mesh/polygon data of its own. This is essentially what can be achieved using a kD-tree -- choosing a split plane at any given node that is positioned such that it splits any remaining geometry into exactly two pieces, even if all that geometry is located on one side of the parent nodes volume.

> **Note:** One possible method for choosing a split plane is to divide the polygon data into two equal sets at a given node. If splitting along the X axis at a given node for example, we could sort all the vertex data by their X component and then build a plane that passes through the median vertex.

kD-trees are generally much deeper trees than oct-trees if you intend to get the same level of spatial partitioning because they only divide space into two spaces at each node, so we will usually have to traverse deeper into a kD-tree. However, each level in the tree has fewer child nodes assigned to it than its equivalent level in an oct-tree, so traversal is cheaper at each level. In the above example, we see a quad-tree that has four nodes on its second level and eight nodes on its third level. The kD-tree on the other hand would have only two nodes on its second level and four nodes on its third level. Therefore, the deeper tree traversal required in the kD-tree case is offset by the speed at which the kD-tree can traverse its tree (to some extent). For example, we may find when testing a bounding volume against the split plane stored in the root that the query volume is contained in the front halfspace of the clip plane. With one plane/volume test we have just rejected half of the entire scene. In an oct-tree, each AABB/AABB test we perform will reject only $1/8^{th}$ of the parent node's volume.

Choosing a split plane at a kD-tree node is not simply about finding the plane that best balances the tree since we sometimes have other considerations (perhaps more important ones) that we wish to factor in. For example, a common goal when choosing a split plane is to find one that produces the least number of polygons that will have to be split (if we are building a clipped tree) in order to fit into the child nodes. Every time we split a polygon into two pieces we increase the polygon count of our scene and this could grow significantly if split planes are being chosen arbitrarily. Furthermore, clipping polygons arbitrarily also often introduces a nasty visual artifact called a T-junction in the polygon data. We will discuss T-junctions later, but for now just know that due to the rounding errors accumulated in the rendering pipeline, two polygons that are neighbors (i.e., they look like one polygon), but that do not have their vertices in identical places along the shared edge, can produce a sparkling effect. Essentially, because their vertices are not in identical places, there are sub-pixel gaps produced during rendering. This results in odd pixels between the two polygons (along the seam) that do not get rendered and anything rendered behind it can show through. You have probably seen this artifact in some video games -- it is commonly referred to as *sparkling*. T-junctions also wreak havoc with vertex lighting schemes as we will find out later.

T-junctions often occur whenever a great deal of polygon clipping is employed on a scene, so these artifacts can manifest themselves with all the tree types we have discussed thus far (or indeed with any clipping process). However, when we are using uniform partitioning (such as with the quad-tree or oct-tree) we generally create many fewer T-junctions in our geometry (this will make sense later when we

cover T-junction repair). When using a kD-tree that is allowed to arbitrarily position its split plane, T-junctions artefacts are common. Although T-junctions can be repaired after the tree has been compiled (we will cover how to do this later in the lesson) the repair of a single T-junction introduces an additional triangle into the scene. Therefore, in the case of a kD-tree that is using non-uniform split planes, hundreds or maybe thousands of T-junctions could be produced by the tree building process. The repair of these T-junctions would introduce hundreds or thousands of additional triangles in our scene, which is far from ideal. The existence of T-junctions are only a concern if the polygon data contained in the tree is intended to be collected and rendered. If the tree polygon data is only being used for collision queries, we can typically leave the T-junctions alone since they will not affect our collision queries.

Because our tree class will also be used as a render tree in addition to a collision query tree, we will force our kD-tree to always split into two equally sized child volumes at any given node. This will hopefully reduce the number of T-junctions produced during the building of the tree and keep our polygon count from growing too large during the T-junction repair.

> **Note**: Do not worry too much about what T-junctions are for now as they will be discussed over the next few sections. They are only mentioned now as a justification for why our kD-tree implementation will always use a split plane at each node which partitions the node's volume into two uniformly sized children.

So the balance of a tree is important but as noted, it is not our only goal when compiling our trees. Often trial and error will produce the best results and this is where benchmarking your code is ***very important***. You may find for example that using an arbitrary split plane at a kD-tree node creates a more balanced tree at the cost of many triangles being inserted into the scene to repair the T-junctions produced. This increase in polygons could create a consistently deeper tree and undo much of what you were trying to achieve by balancing the tree in the first place.

We have now discussed the theory behind the oct-tree, the quad-tree, and the kD-tree and we are ready to start looking at the code to both the tree building processes as well as the support structures, routines, and intersection queries that will need to be implemented. We will delay our discussion of the BSP tree until Chapter 16, after we have stepped through the source code needed to build the three tree types we have already discussed. We will also discuss some of the core mathematical routines we will need to run queries on our tree. The BSP tree, although constructed in a similar way, ultimately exhibits very different characteristics due to the fact that it divides space arbitrarily. The application of a BSP tree is often done to achieve a different goal than just spatial subdivision as we will see later in the course.

In the remainder of this textbook, we will cover all the code that builds the quad-tree, oct-tree, and kD-tree. Ultimately we will integrate them into our collision system as a broad phase suite. Building a tree rendering system that is hardware friendly is rather complex and will be discussed in its own chapter. Lab Project 14.1 uses the rendering system that will be discussed in the following lesson, so you should probably ignore most of the tree rendering code for now.

14.9 Polygon Clipping

Several times throughout this chapter we have mentioned that our system will have the optional ability to create clipped trees. That is, trees that are constructed such that the polygon data stored in each leaf fits completely in its volume. This means that the tree building process will involve a good amount of clipping of the original polygon data to the split planes that subdivide the node into its children. This section will discuss the process of clipping polygons.

If you have never implemented code that clips a polygon to a plane, you will probably be pleased to learn how simple it actually is. The basic process is one of stepping around each edge of the polygon and classifying its vertices against the plane in order to build two vertex lists that will describe the front and back split polygons (i.e., the polygon fragments that lay on each side of the plane). If the current vertex being processed is in front of the plane and the following vertex is also in front of (or on) the plane, then the current vertex belongs in the front split polygon and is added to its vertex list. Likewise, if the current vertex being processed is behind the plane and the next vertex in the list is also either behind or on the plane, then the current vertex being processed is added to the vertex list of the back split polygon.

If however, the current vertex being processed and the next vertex in the list are on opposing sides of the plane, then we have found an edge in the original polygon where an intersection occurs with the plane. In this instance we use the two vertices in the edge to create a ray and perform a Ray/Plane intersection test with the split plane. The result of this intersection test will be the position on the plane at which the ray intersects it. The vertex in the edge that was in front of the plane is added to the front split polygon and the other vertex in the edge located on the back side of the plane is added to the back split polygon. The intersection point where the ray intersected the plane is made into a new vertex and is added to *both* the front split and the back split polygons.

After having done this for every vertex/edge in the original polygon, will have two new polygons and the original polygon can be discarded. Obviously, if you do not intend to the split the polygon into two fragments, but are instead interested only in clipping polygons to a plane, you can simply discard the vertices on the half space of the plane that you are clipping away and return only a single polygon. Thus, a polygon splitting function could easily double as a polygon clipping function by simply being instructed to discard any vertices located in a certain plane halfspace. We do a similar thing in our polygon clipping function. The function will accept as parameters the original polygon that is to be split/clipped and will also be passed pointers to two polygons that, on function return, will be filled with the front split and back split polygons. Passing NULL as one of these parameters will turn the splitter function into a clipper function. If you pass NULL as the parameter where you would normally pass a pointer to a polygon that will receive the back split data, the function will recognize that you have no interest in the vertex data that is behind the plane and simply clip the polygon to the plane and return only the section of the polygon that is in the plane’s front halfspace.

Figure 14.44 shows how a pentagonal polygon would be split into two by an arbitrary split plane.

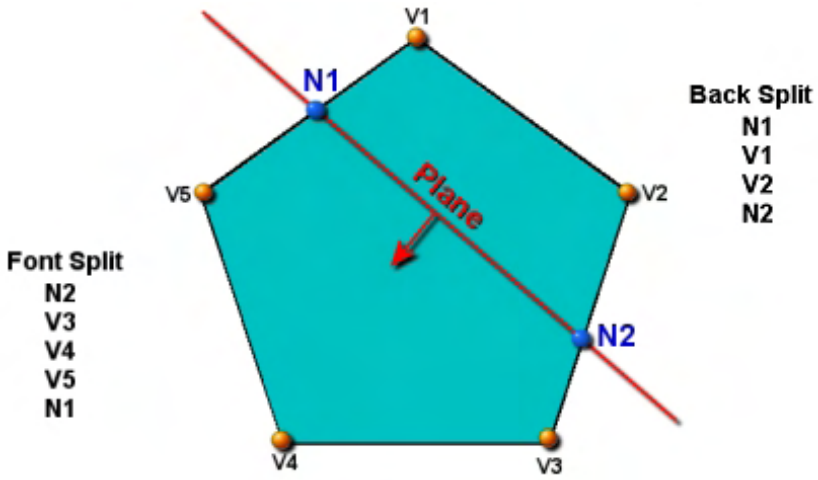


Figure 14.44

Let us step through this example using the technique described above. The original polygon has five vertices labelled v1 – v5 as shown. The plane can be seen as the red line in the diagram clearly carving the polygon into two parts. We would set up a loop starting at the first vertex v1. We would classify this point against the plane and discover it is in the back space of the plane. If v2 is also in the plane's back space, then a new polygon structure is allocated for the back split and v1 is added to its vertex list.

In the next loop iteration we check vertex v2 which is also in the plane's back space. If v3 is also in the plane's back space, then v2 would be added to the back split polygon also. However in this case we can see that v3 is actually in the plane's front space so we know for sure that the polygon edge (v2,v3) intersects the plane. Therefore, we create a ray from v2 to v3 and perform a ray/plane intersection test. This will return the position at which the ray intersects the plane, shown as N2 in the diagram. We add v2 and N2 to the vertex list of the back split polygon and add the vertex N2 to the vertex list of the front split polygon as well. The back split polygon now has vertices v1,v2, and N2 in its vertex list and the front split polygon so far has a single vertex, N2.

Next we test vertex v3 and find it is in the plane's front space. Because v4 is also in the front space, v3 is added to the vertex list of the front split polygon so that now it contains two vertices, N2 and v3. Next we test vertex v4 which is in the plane's front space also. v4 is added to the front split polygon's vertex list because the next vertex v5 is also in front of the plane. The front split polygon now has vertices n2, v3, and v4 in its vertex list.

Finally, in the last iteration of the loop we classify vertex v5 against the plane and discover it is in the plane's front space. However, vertex v5 is the last vertex in the original polygon and it forms an edge with the first vertex v1, so we must test that this edge does not span the plane. We discover that v1 is in the back space of the plane and therefore the edge (v5 ,v0) does indeed span the plane. Thus, we create a ray from v5 to v1 and perform a ray/plane intersection calculation to return to us the intersection point N1 where the ray intersects the plane. v5 and N1 are added to the front split polygon and N1 is also added to the back split polygon and we are done. In this example, the front split polygon would have vertices N2, v3, v4, v5, and N1. The back split polygon would have vertices v1, v2, N1, and N2. We now have two polygons with a clockwise winding order which can be returned from our polygon splitting function. At this point, the original polygon can be discarded.

14.10 Implementing Hierarchical Spatial Partitioning

In order to split and clip polygons, certain utility functions will be added to our CCollision class as static methods. We will add methods to help our clipping routines as well as new bounding volume intersection methods that will be used by the tree during traversals. The CCollision class has very much become our core library of intersection and plane classification routines.

When discussing the process of polygon splitting in the last section it became apparent that much of the process involves the classification of each vertex in the source polygon against the split plane. This is so we can determine whether a given vertex belongs to the front split or the back split fragment. Although we have discussed several times in past lessons how the classification of a point against a plane is a simple dot product and an addition in order to compute the plane equation, we have wrapped this operation in a method called PointClassifyPlane in our CCollision object. Instead of just returning the result of the dot product, the function returns one of four possible members of the CLASSIFYTYPE enumerator shown below (defined in the CCollision namespace):

Excerpt from CCollision.h

```
// Enumerators
enum CLASSIFYTYPE { CLASSIFY_ONPLANE = 0,
                    CLASSIFY_BEHIND = 1,
                    CLASSIFY_INFRONT = 2,
                    CLASSIFY_SPANNING = 3 };
```

As you can see, it is much nicer during our building routines to get back a result that clearly is behind, in front, on plane, or spanning the plane rather than us having to determine this ourselves using the sign of the dot product result. The PointClassifyPlane function will only ever return either CLASSIFY_ONPLANE, CLASSIFY_BEHIND or CLASSIFY_INFRONT as it is impossible for a single point in space to be spanning a plane. However, the CLASSIFY_SPANNING member will be returned by other classification functions that we will discuss shortly. Let us now look at this new function (which is really just a dot product wrapper).

14.10.1 PointClassifyPlane – CCollision (static)

The PointClassifyPlane method accepts three parameters. The first is the vector describing the point we would like to classify with respect to the plane, the second is the normal of the plane, and the third is the distance to the plane from the coordinate system origin. The function then evaluates the plane equation by performing the dot product between the point and the plane normal (two 3D vectors) and then adding the resulting distance. This is obviously equivalent to performing AX+BY+CZ+D where ABC is the plane normal, D is the plane distance, and XYZ is the point being classified. The result of the plane equation is stored in the local variable fDistance.

```
CCollision::CLASSIFYTYPE CCollision::PointClassifyPlane
                         ( const D3DXVECTOR3& Point,
                           const D3DXVECTOR3 &PlaneNormal,
```

```
                            float PlaneDistance )
{
    // Calculate distance from plane
    float fDistance = D3DXVec3Dot( &Point, &PlaneNormal ) + PlaneDistance;

    // Retrieve classification
    CLASSIFYTYPE Location = CLASSIFY_ONPLANE;
    if ( fDistance < -1e-3f ) Location = CLASSIFY_BEHIND;
    if ( fDistance >  1e-3f ) Location = CLASSIFY_INFRONT;

    // Return the classification
    return Location;
}
```

Be default we assume that the point is on the plane by setting the local Location variable to CLASSIFY_ONPLANE and then go on to test the sign of the result of the plane equation. If fDistance is less than zero (with tolerance) then the point is behind the plane. If fDistance is greater than zero then the point is in the plane's front halfspace. At the bottom of the function the local Location variable will contain either CLASSIFY_ONPLANE, CLASSIFY_BEHIND or CLASSIFY_INFRONT which is then returned to the caller.

The above function will be used by our polygon splitting routine, but the following routine will be used by the core tree building functions to determine if a given polygon needs to be split. It is this next routine that can be used to determine whether a given polygon is in front or behind a plane, lying on the plane, or spanning the plane. In the spanning case we know we have a polygon that has vertices on both sides of the plane, which tells us the polygon needs to be split.

14.10.2 PolyClassifyPlane – CCollision (static)

This is another very useful function that you will likely find yourself using many times as you progress in your 3D programming career. For this reason, we have not built it into the tree code, but have added it to our collision library where we can continue to use it in the future without any dependency on anything else. Remember, these static members always exist even when an instance of the CCollision object does not. Thus CCollision represents a handy collection of intersection routines that can be used by the application even if the application is not using the actual collision system.

This function has a fairly simple task in that it is really just performing the test described in the above function for each vertex in the passed polygon. That is, we classify each point in the polygon against the plane and keep a record of how many points were found behind the plane, in front of the plane, and on the plane. After testing each vertex we can then examine these results to get our final outcome. For example, if the number of vertices found in front of the plane is equal to the vertex count of the polygon, then it must mean the entire polygon is in front of the plane. Likewise, if the number of vertices found behind the plane is equal to the vertex count of the passed polygon then the entire polygon must lay behind the plane. If all points lay on the plane then the polygon is obviously on the plane and we return that result. Finally, we know that if none of the above conditions are true then it must mean some vertices were behind the plane and some were in front of it and therefore, we return CLASSIFY_SPANNING.

Although our spatial trees will all store their static polygon data at the leaves of the tree in CPolygon structures (something we will look at in a moment) we really do not want any generic and potentially reusable routine that we add to our collision namespace to be so reliant on external structures. For example, in a future lab project we might not store our polygon data in a CPolygon object, or we may change the format of its internal vertices, which would mean in either case that we would not be able to use this function. Therefore, we have kept this function as generic as possible by allowing the application to pass the vertex data of the polygon being classified as three parameters.

The first parameter is a void pointer which should point to an array of vertices which define the polygon. This is a void pointer so that it can be used to point to any arbitrary array of vertex components. For example, this could be a pointer into a locked system memory vertex buffer or the vertex array of a CPolygon structure. As this function is only interested in the positional data of each vertex and has no interest in other vertex components, we could also just pass an array of 3D position vectors. The second parameter is the number of vertices in this array so that the function knows how many vertices it has to test. The third parameter is a stride value describing the size of each vertex in the array (in bytes). Remember, this function has no idea how many vertex components you have in your structure, so it has no idea how large each one is and how many bytes a pointer must be advanced to point to the next vertex in the array. Passing a stride value solves this problem and allows the function to step from vertex to vertex in the array without caring about what other data follows the positional data. As the fourth and fifth parameters we pass the normal and the distance of the plane which we wish to classify the polygon against. Below we present the complete code listing to the function, which you should be able to understand with very little explanation.

```
CCollision::CLASSIFYTYPE CCollision::PolyClassifyPlane(
                                                void *pVertices,
                                                ULONG VertexCount,
                                                ULONG Stride,
                                                const D3DXVECTOR3& PlaneNormal,
                                                float PlaneDistance )
{
    ULONG   Infront  = 0, Behind = 0, OnPlane=0, i;
    UCHAR   Location = 0;
    float   Result   = 0;
    UCHAR  *pBuffer  = (UCHAR*)pVertices;

    // Loop round each vector
    for ( i = 0; i < VertexCount; ++i )
    {
        // Calculate distance from plane
        float fDistance = D3DXVec3Dot((D3DXVECTOR3*)pBuffer,
                                      &PlaneNormal ) + PlaneDistance;
        pBuffer += Stride;

        // Retrieve classification
        Location = CLASSIFY_ONPLANE;
        if ( fDistance < -1e-3f ) Location = CLASSIFY_BEHIND;
        if ( fDistance >  1e-3f ) Location = CLASSIFY_INFRONT;

        // Check the position
        if (Location == CLASSIFY_INFRONT )
            Infront++;
        else if (Location == CLASSIFY_BEHIND )
```

```
            Behind++;
        else
        {
            OnPlane++;
            Infront++;
            Behind++;

        } // End if on plane

    } // Next Vertex

    // Return Result
    if ( OnPlane == VertexCount ) return CLASSIFY_ONPLANE;     // On Plane
    if ( Behind  == VertexCount ) return CLASSIFY_BEHIND;      // Behind
    if ( Infront == VertexCount ) return CLASSIFY_INFRONT;     // In Front
    return CLASSIFY_SPANNING; // Spanning
}
```

In the above code we first cast the void pointer to a byte pointer so that the stride value will describe exactly how much we need to increment this pointer to step to the next vertex. We then set up a loop to loop through each vertex of the polygon. For each vertex we compute the plane equation for that vertex and the plane and store the result in fDistance. We then increment the vertex pointer by the stride value so that it is pointing at the next vertex in the array. Next we test the value of fDistance and set the value of Location to either CLASSIFY_ONPLANE, CLASSIFY_INFRONT or CLASSIFY_BEHIND depending on the result. Then we test the value of the Location variable and increment the relative counter. For example, if the vertex is behind the plane, then we increment the Behind counter. If it is in front of the plane we increment the InFront counter. Otherwise it means the vertex is on the plane and we increment the OnPlane, InFront, and Behind counters. It is very important that in the on plane case we also increment the InFront and Behind counters because these counters are trying to record how many vertices would exist in a front split and back split polygon were we actually creating one. Obviously, if a point is on the plane, it could belong to either a front split or a back split polygon. Using this strategy allows us to perform simple tests at the bottom of the function to determine if the polygon is considered in front or behind the plane. Remember, a polygon may still be considered to be behind a plane even if some of its vertices are touching the plane.

At the bottom of the main for loop we can see that we have stored the values of how many vertices to be found in each condition in the OnPlane, Behind and InFront local variables. If the OnPlane value is equal to the vertex count for example, then we know the polygon is completely on the plane. If the vertex count is equal to the number vertices found to be in front or behind the plane then it must mean the polygon lay in front or behind the plane, respectively. Finally, if none of these cases are true we return CLASSIFY_SPANNING since there were obviously vertices found in both plane halfspaces.

14.10.3 CPolygon – Our Static Geometry Container

Earlier we noted that our spatial tree objects will store static polygon data in the leaves of the tree. Therefore, we need a data structure that we can use to pass this data into our tree (prior to the build process being called). This same structure will also be used once the tree is compiled to store the polygon data at the leaf nodes. The object we use for this is defined in CObject.h and CObject.cpp and is called CPolygon.

CPolygon is a simple class that essentially just contains a list of CVertex structures and has two methods that allow us to add vertices to its list or insert vertices in the middle of its list. We will not be showing the code to these functions as they are simple array resize and manipulation functions the likes of which we have seen many times before. However, this object also has a method called Split which allows us to pass in a plane and it will clip/split the polygon to that plane. The CPolygon object's vertex array is not altered by this process as this function allocates and returns two new CPolygon objects containing the front and back split fragments. After calling this function and retrieving the two split polygon fragments the caller will usually delete the original polygon as it is probably no longer needed. Below we show the class declaration in CObject.h.

```
class CPolygon
{
public:

    // Constructors & Destructors for This Class.
             CPolygon();
    virtual ~CPolygon();

    // Public Functions for This Class
    long            AddVertex           ( USHORT Count = 1 );
    long            InsertVertex        ( USHORT nVertexPos );
    bool            Split               ( const D3DXPLANE& Plane,
                                          CPolygon ** FrontSplit = NULL,
                                          CPolygon ** BackSplit = NULL,
                                          bool bReturnNoSplit = false ) const;

    // Public Variables for This Class
    ULONG       m_nAttribID;            // Attribute ID of face
    D3DXVECTOR3 m_vecNormal;            // The face normal.
    USHORT      m_nVertexCount;         // Number of vertices stored.
    CVertex    *m_pVertex;              // Simple vertex array
    D3DXVECTOR3 m_vecBoundsMin;    // Minimum bounding box extents of this polygon
    D3DXVECTOR3 m_vecBoundsMax;    // Maximum bounding box extents of this polygon
    ULONG       m_nAppCounter;     // Automatic 'Already Processed' functionality
    BOOL        m_bVisible;        // Should it be rendered
};
```

When our application loads static geometry from an IWF file (inside CScene::ProcessMeshes) we will store each face we load in a new CPolygon object and pass it to the spatial tree for storage. After the spatial tree's Build function has been called, the tree will contain a number of leaf structures, each containing an array of the CPolygon pointers that exist in that leaf. A CPolygon structure will contain a convex clockwise winding N-gon which means a single polygon may represent multiple triangles.

The member variables are fairly self-explanatory but we will briefly explain them here since some new ones have been added since the introduction of this class in some of our Module I lab projects.

ULONG m_nAttribID
Each polygon will store an attribute ID that has some meaning to our application (e.g., which textures and materials will need to be set when this polygon is rendered). Therefore, when we create a CPolygon object during the loading of an IWF file, we will store the global subset ID of the polygon in this member. As with our previous lab projects, this will actually be an index into the scene's attribute array where each element in that array describes the texture and material that should be bound to the device prior to rendering this polygon. Although we will not discuss our rendering strategy until the next lesson, this member is used by the tree so that it can render all polygons from all visible leaves together so that efficient batch rendering is maintained.

D3DXVECTOR3 m_vecNormal
This member will contain the normal of the polygon.

USHORT m_nVertexCount
This member contains the number of vertices currently stored in this polygon and contained in the m_pVertex array described next.

CVertex *m_pVertex
This is a pointer to the polygon's vertex array. Each element in this array is the now familiar CVertex object, which contains the positional data of the vertex, its normal, and its texture coordinates.

D3DXVECTOR3 m_vecBoundsMin
D3DXVECTOR3 m_vecBoundsMax
In these two members we will store the world space axis aligned bounding box of the polygon. That is, each polygon in our spatial tree will contain an AABB that will be used to speed up collision testing against the spatial tree's geometry database.

Our collision system will now store its static geometry in a spatial tree instead of just in a polygon array. When collision queries are performed, the collision system will ask the spatial tree for a list of all the leaves that the swept sphere's AABB intersects (broad phase). Once the collision system is returned a list of leaves that contain the potential colliders, we could just collect the polygons from each returned leaf and send them to the narrow phase process. However, the actual intersection tests performed between the swept sphere and the polygon are very expensive, and even though we have rejected a large number of polygons by only fetching polygon data from the leaves that intersect the swept sphere AABB, many of the polygons contained in these leaves may be positioned well outside this AABB. Therefore, once we have the list of leaves which the swept sphere's AABB intersects, we will loop through each polygon in those leaves and test its bounding box against the bounding box of the swept sphere. This way we avoid transforming polygons into eSpace and performing the full spectrum of intersection tests on it when we can quickly tell beforehand that the polygon and the sphere could not possibly intersect because their AABBs do not intersect. Although this might sound like a small optimization to the broad phase which would hardly seem worth the memory taken up by storing an AABB in each polygon, the speed improvements to the broad phase on our test machines were very

dramatic. Using the spatial tree and this additional broad phase step in our collision system, queries on large scenes increased in performance by a significant amount.

> **Note:** We tested our collision system using a fairly large level Quake III™ level (on a relatively low end machine) prior to adding the broad phase. Due to the fact that the collision system had to query the entire scene each frame at the polygon level, a collision query was taking somewhere in the region of 10 to 12 seconds. With the spatial tree added, the time was reduced to a few milliseconds, allowing us to achieve interactive frame rates well above 60 frames per second. What a difference, 60 frames per second versus 1 frame every 12 seconds. Even on a simple level such as colony5.iwf, frame rate jumped from 30 frames per second to over 300 frames per second with the introduction of the broad phase. Proof for sure that the broad phase component of any collision system is vitally important. Also proof that hierarchical spatial partitioning is an efficient way to get access to only the areas of the scene you are interested in for a given query.

ULONG m_nAppCounter

This member is used to avoid testing a polygon for collision multiple times if a non-clipped spatial tree is being used. As we know, if a non-clipped tree is constructed, a single polygon may span multiple leaves. This means its CPolygon pointer will be stored in the polygon array of multiple leaf structures. When the collision system fetches the list of intersecting leaves from the spatial tree it has to make sure that if a polygon exists in multiple leaves it is not tested twice.

To get around this problem we decide to add an application counter member variable to our CGameApp class. This is a simple DWORD value that can be incremented via the CGameApp::IncrementAppCounter method and retrieved via the CGameApp::GetAppCounter. In other words, this is a simple value that can be increment by external components.

Our collision system uses the app timer in the following way: Prior to a collision test, the app counter is incremented and then fetched so that we now have a new unique counter value for this update. When we fetch the intersecting leaves from the spatial tree and find a polygon that needs to be tested at the narrow phase level, we will store the app counter value in this member of the CPolygon. If a little later, when testing the polygons from another leaf, we find that we are about to test a polygon that has an m_nAppCounter value which is equal to the current value of the CGameApp's application counter, it must mean that this polygon has already been processed in this update because it belonged in a leaf that we have previously tested. Therefore, we do not need to test it again.

This is a much nicer and more efficient solution than simply storing a 'HasBeenTested' boolean in the polygon structure because this would involve us having to reset them all back to false prior to performing another collision test. We would certainly not want to have to do that for every polygon in the scene. By storing the application counter in the polygon, as soon as we wish to perform another query, we can just increment the application timer which will immediately invalidate all the polygons because their m_nAppCounter variables will no longer match the current value of the application counter.

bool m_bVisible
We will see this member of the polygon structure being used in the following lesson when we implement the rendering system. Essentially, it allows us to flag a polygon as being invisible when added to the tree. This is useful if we wish to add a polygon to the collision system but would not like to have it rendered The collision system will ignore this member and will test the swept sphere against any polygons in its vicinity. The rendering system however (which will use the same tree) will only render a polygon if it has this boolean set to true. This boolean is set to true by default in the CPolygon constructor.

Split - CPolygon

The only method of CPolygon that we have to discuss is an important one since it is the key to creating a clipped tree. The CPolygon::Split method implements the splitting strategy discussed in the last section. Unfortunately, the code may at first seem a little confusing due to two reasons. First, we want the function to also double as a clipper, so there are several conditional blocks in the code that only get executed if the caller has requested that it is interested in getting back the relevant split fragment. For example, if the caller passes NULL as the FrontSplit parameter the function will discard any portion of the polygon that lies in front of the plane and will only return the back split fragment. Second, the function is ordered in a way that means we only allocate the memory for the new polygon data when we know we actually need it.

We have also added a fourth parameter to this function, which is a boolean that allows us to specify what should happen when the passed plane does not intersect the polygon. When this boolean is set to false, the polygon data will always be created and returned in either the front or back split. That is, if the polygon is completely in front of the passed plane, a new front split polygon will be returned which contains a copy of all the data from the original polygon. This way we will always get back either a front split polygon or a back split polygon even if the polygon is not spanning the plane. Although this might sound like a strange way for the function to behave, it can be useful during a recursive clipping process to always assume that the original polygon is no longer needed after its Split method has been called. However, it is also often the case that if the polygon does not span the passed plane, then you do not want any front split or back split polygon created and would rather just continue to work with the original polygon. This is what the final parameter to this function is for. If set to true, it will only return front split and back split polygons if the original polygon is spanning the plane. If not, it will just return immediately and essentially will have done nothing. If set to false, it will always return a polygon in either the front or back split polygons even if the polygon was completely in front or completely behind the plane. In other words, the function will always create a new polygon in this instance allowing you to discard the original one.

Let us have a look at the code to this function a section at a time. There are four parameters. The first is the clip/split plane and it is expected in the form of a D3DXPLANE structure which describes the plane in AX+BY+CZ+D format. The second and third parameters are the addresses of CPolygon pointers which on function return will point to the new front and/or back polygon fragments. The caller does not have to allocate the CPolygon objects to contain the front and back splits, it simply has to pass the address of two CPolygon pointers. The Split function will allocate the CPolygon objects and assign the pointers you pass to point at them, so the caller can access them on function return. The fourth parameter

is the boolean parameter describing whether the function should only create new polygons if the polygon is spanning the plane or whether it should do nothing.

```
bool CPolygon::Split( const D3DXPLANE& Plane,
                      CPolygon ** FrontSplit,
                      CPolygon ** BackSplit,
                      bool bReturnNoSplit ) const
{
    CVertex    *FrontList     = NULL, *BackList = NULL;
    USHORT      CurrentVertex = 0, CurrentIndex = 0, i = 0;
    USHORT      InFront       = 0, Behind       = 0, OnPlane = 0;
    USHORT      FrontCounter  = 0, BackCounter  = 0;
    UCHAR      *PointLocation = NULL, Location;
    float       fDelta;

    // Bail if no fragments passed (No-Op).
    if (!FrontSplit && !BackSplit) return true;

    // Separate out plane components
    D3DXVECTOR3 vecPlaneNormal = (D3DXVECTOR3&)Plane;
    D3DXVECTOR3 vecPlanePoint  = vecPlaneNormal * -Plane.d;
```

The first section of the function is fairly simple. First, if the caller has not passed a front split or a back split pointer then we have no way of returning any clipped/split polygon data back anyway so we may as well just immediately return. It is ok to pass only one pointer (either a front split pointer or a back split pointer) which will turn the function into a clipper instead of a splitter. However, if no pointers are passed, there is no point continuing.

We can also see how we use the passed D3DXPlane structure to separate the plane into two components, a plane normal and a point on that plane. Getting the plane normal is easy as the first three members of the plane structure (a, b, and c) are the plane normal. So we can do a straight cast of these three members into a 3D vector. Calculating a point on the plane is also easy as we know that the **d** member contains the distance to the plane from the origin. That means that if we moved a point from the origin of the coordinate system in the direction of the plane normal for this distance we would have a point on that plane. As we are using the AX+BY+CZ+D version of the plane equation, D is a negative value for points behind the plane. Therefore, we just have to negate its sign and use it to scale the normal and we will have created that point on the plane.

In the next section of code we prepare the memory that will be needed for performing the split operation and the various tests. First we will need to know the location of each vertex in the polygon with respect to the plane, so we will allocate an array of unsigned chars (one for each vertex in the original polygon). In a moment, we will loop through each vertex and use the CCollision::PointClassifyPlane method we implemented earlier. This will return either CLASSIFY_FRONT, CLASSIFY_BACK or CLASSIFY_ONPLANE. We will store the results in the array.

```
    try
    {
        // Allocate temp buffers for split operation
        PointLocation = new UCHAR[m_nVertexCount];
        if (!PointLocation) throw std::bad_alloc(); // VC++ Compat
```

As you can see, this array is called PointLocation and it must be large enough to hold the classification result of each vertex in the original polygon. In a moment we will fill this array with the classification results of each vertex with respect to the plane.

What we will do now is also allocate two arrays of vertices. These two arrays will be used to collect the vertices that get added to the front split and back split polygons respectively. Of course, if the caller did not pass in either a front split or a back split pointer, then it means we do not have to allocate a vertex array to deal with vertices in that halfspace of the plane. That is, if only the address of a CPolygon pointer has been passed for the FrontSplit parameter, there is no need to allocate a vertex array to collect vertices that are in the back split. Also, if the user as passed true as the bReturnNoSplit parameter, then it means this function should only create a front split and back split polygon if the polygon is spanning the plane. Therefore, if true has been passed, we will not allocate those vertex arrays here; we will allocate them later when we know we definitely have a split case. It would be wasteful to allocate the memory for either a front split or a back split polygon only to find that we do not need it because the caller has requested that no polygons be created in the no split case.

```
        // Allocate these only if we need them
        if (FrontSplit && !bReturnNoSplit)
        {
            FrontList = new CVertex[m_nVertexCount + 1];
            if (!FrontList) throw std::bad_alloc(); // VC++ Compat
        } // End If

        if ( BackSplit && !bReturnNoSplit )
        {
            BackList = new CVertex[m_nVertexCount + 1];
            if (!BackList) throw std::bad_alloc(); // VC++ Compat
        } // End If

    } // End Try

    catch (...)
    {
        // Catch any bad allocation exceptions
        if (FrontList)      delete []FrontList;
        if (BackList)       delete []BackList;
        if (PointLocation)  delete []PointLocation;
        return false;
    } // End Catch
```

Notice that when we allocate the front and back split vertex arrays we make them large enough to hold one more than the number of vertices in the original polygon that is being clipped.

We do this because when we clip or split a polygon to a plane, although new vertices are introduced along the edges that intersect the plane (for both fragments), none of the fragments can ever have its vertex count increased beyond the original number of vertices plus one. This is shown in Figure 14.45 where a triangle in split to an arbitrary plane creating two polygons. One has four vertices and the other has three. It should be noted that this rule is only applicable to convex polygons. Since we are always dealing with convex polygons this is just fine.

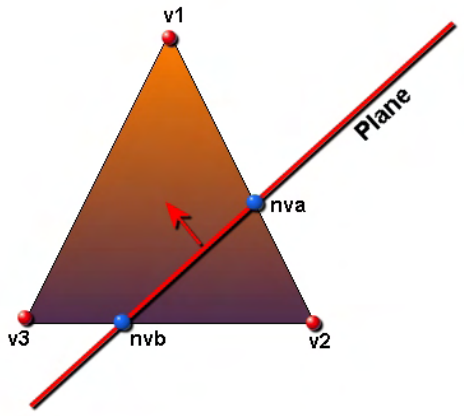


Figure 14.45

In this image we can see that the triangle to be clipped originally consisted of vertices v1, v2, and v3. However, during the testing of the edges, two were found to be spanning the plane and the vertices nva and nvb are generated and added to both the front and back vertex lists. This generates a front split vertex list of v1, nva, nvb and v3 and a back split vertex list of nva, v2 and nvb. No matter how we rotate the plane and regardless of the shape of the original polygon being clipped, we will never generate a new split polygon that has more than one additional vertex beyond the original polygon from which it was clipped/split from. So we can see in Figure 14.45 that if we allocate both the front and the back vertex lists to hold four vertices (number of vertices in the original triangle plus one) we definitely have enough room to store the vertices for each fragment.

At this point we have front and back vertex lists allocated (only if bReturnNoSplit equals false) and we also have a PointLocation array large enough to child a classification result for each vertex in the original polygon. Let us now fill out the point location array by looping through each vertex in the polygon and classifying it against the plane (using our new PointClassifyPlane function). We store the result for each vertex in the corresponding element in the PointLocation array as shown below.

```
    // Determine each points location relative to the plane.
    for ( i = 0; i < m_nVertexCount; ++i )
    {
        // Retrieve classification
        Location = CCollision::PointClassifyPlane( (D3DXVECTOR3&)m_pVertex[i],
                                                   (D3DXVECTOR3&)Plane,
                                                    Plane.d );

        // Classify the location of the point
        if (Location == CCollision::CLASSIFY_INFRONT ) InFront++;
        else
        if (Location == CCollision::CLASSIFY_BEHIND ) Behind++;
        else
        OnPlane++;

        // Store location
        PointLocation[i] = Location;

    } // Next Vertex
```

Now we have an array that stores whether each vertex is in front, behind, or on the plane. Note that we maintain three counters (InFront, Behind, or OnPlane) so that at the end of the above loop we know exactly how many vertices (if any) are in each classification pool.

In the following code section we test the value of the InFront counter and if it is not greater than zero then it means no vertices were located in the front halfspace. If the caller passed true for bReturnNoSplit, then there is nothing to do other than release the PointLocation array and return. This is because if the InFront counter is set to zero, the polygon cannot possibly span the plane. Thus in the bReturnNoSplit=true case, this means we just wish the function to return.

```
    // If there are no vertices in front of the plane
    if (!InFront )
    {
        if ( bReturnNoSplit ) { delete []PointLocation; return true; }
        if ( BackList )
        {
            memcpy(BackList, m_pVertex, m_nVertexCount * sizeof(CVertex));
            BackCounter = m_nVertexCount;
        } // End if

    } // End if none in front
```

Notice that if the InFront counter is zero but the caller did not pass true as the bReturnNoSplit parameter, all the vertices in the polygon belong to the back split fragment. When this is the case we copy all the vertex data from the source polygon into the vertex list being compiled for the back split. We also set the BackCounter variable equal to the number of vertices in the source polygon so we know how many vertices we have collected in the BackList vertex array.

In the next snippet of code we do exactly the same thing again only this time we are handling the case where no vertices were found to lie behind the plane. When this is the case we simply return if bReturnNoSplit was set to true, or we consider all the vertices of the source polygon to belong to the back split polygon and copy over its vertices into the front split vertex list.

```
    // If there are no vertices behind the plane
    if (!Behind )
    {
        if ( bReturnNoSplit ) { delete []PointLocation; return true; }
        if ( FrontList )
        {
            memcpy(FrontList, m_pVertex, m_nVertexCount * sizeof(CVertex));
            FrontCounter = m_nVertexCount;
        } // End if

    } // End if none behind
```

If both the InFront and Behind counters are set to zero then it must mean that every vertex in the source polygon lies on the split plane. This means no splitting is going to occur and the polygon does not belong in any halfspace. When this is the case we return.

```
    // All were onplane
    if (!InFront && !Behind && bReturnNoSplit )
       { delete []PointLocation; return true; }
```

Earlier in the function we allocated arrays to hold the front and back vertex lists but only if the bReturnNoSplit parameter was set to false. This is because one way or another we knew we would definitely need them even if the polygon was not spanning the plane because the function will always return a copy of the source polygon in either the front or back split polygons. However, we did not allocate these arrays if the bReturnNoSplit boolean was set to true since we would only need the front and back split vertex lists compiled if the source polygon is spanning the plane. Otherwise the function would have simply returned by now and done nothing.

In the next section of code we perform the front and back vertex list allocations for the bReturnNoSplit is true case. The memory will have already been allocated for these arrays earlier in the function if bReturnNoSplit was false.

```
    // We can allocate memory here if we wanted to return when no split occurred
    if ( bReturnNoSplit )
    {
        try
        {
            // Allocate these only if we need them
            if (FrontSplit)
            {
                FrontList = new CVertex[m_nVertexCount + 1];
                if (!FrontList) throw std::bad_alloc(); // VC++ Compat
            } // End If

            if ( BackSplit )
            {
                BackList = new CVertex[m_nVertexCount + 1];
                if (!BackList) throw std::bad_alloc(); // VC++ Compat
            } // End If

        } // End Try

        catch (...)
        {
            // Catch any bad allocation exceptions
            if (FrontList)      delete []FrontList;
            if (BackList)       delete []BackList;
            if (PointLocation)  delete []PointLocation;
            return false;
        } // End Catch

    } // End ReturnNoSplit case
```

This is all pretty familiar stuff. We know at this point that if bReturnNoSplit is true that the polygon must be spanning, otherwise we would have returned by now. So, provided the caller passed valid front and back CPolygon pointers, we allocate the vertex arrays for the front and back lists just as we did earlier for the case when bReturnNoSplit did not equal true.

At this point, regardless of the mode of the function being used, we have two empty vertex arrays that will be used to hold the vertex lists for the front and back split polygons. It is now time to look at the core piece of the function that populates the vertex lists for the front and back splits and creates new vertices when an edge is found to be spanning a plane. The code is only executed if the InFront and Behind counters are both non zero since this will only be the case when the polygon is spanning the plane. Remember, earlier in the function, if the polygon was found to be totally on one side of the plane then the source polygon's vertices were copied over into the vertex array for either the front or back split.

The following code loops through each vertex in the polygon and compares the classification result we stored for it earlier with the classification result of the next vertex in the list (the other vertex sharing the edge). If the entire edge is on one side of the plane then the vertex being tested is added to the front or back list. If however, the vertex being tested and the next vertex in the list are on opposing sides of the plane, the edge formed by those two vertices spans the plane and a new vertex is generated on the plane. This new vertex is added to both the front and the back lists.

```
    // Compute the split if there are verts both in front and behind
    if (InFront && Behind)
    {
        for ( i = 0; i < m_nVertexCount; i++)
        {
            // Store Current vertex remembering to MOD with number of vertices.
            CurrentVertex = (i+1) % m_nVertexCount;

            if (PointLocation[i] == CCollision::CLASSIFY_ONPLANE )
            {
                if (FrontList) FrontList[FrontCounter++] = m_pVertex[i];
                if (BackList)  BackList [BackCounter ++] = m_pVertex[i];
                continue; // Skip to next vertex
            } // End if On Plane

            if (PointLocation[i] == CCollision::CLASSIFY_INFRONT )
            {
                if (FrontList) FrontList[FrontCounter++] = m_pVertex[i];
            }
            else
            {
                if (BackList) BackList[BackCounter++] = m_pVertex[i];

            } // End if In front or otherwise
```

In the first section of the loop shown above we set up a loop to iterate through the PointLocation results for each vertex. Loop variable *i* contains the index of the current vertex being processed and local variable CurrentVertex contains the index of the next vertex in the list. Notice we do the modulus operation to make sure that this wraps back around to vertex zero which forms the end vertex for the final edge.

First, we test to see if the classification result for the vertex we are currently testing is CLASSIFY_ONPLANE. If it is, then the vertex itself should be added to both of the vertex lists. On plane vertices will belong to both of the split polygons as shown in Figure 14.46. You can see in the above code that when this is the case, we copy the vertex into both the front and back vertex lists. Notice

that as we do we increment the FrontCounter and BackCounter local variables that are initialized to zero at the head of the function and are incremented in this loop every time we add a vertex to a list. This is so we can keep a count of the number of vertices in each list which we will need at the bottom of the function when we allocate the back split and front split polygons.

If the current vertex is not on the plane then we test to see if it is in front of the plane and if so add the vertex to the front list. Otherwise, the vertex is behind the plane so we add it to the back list.

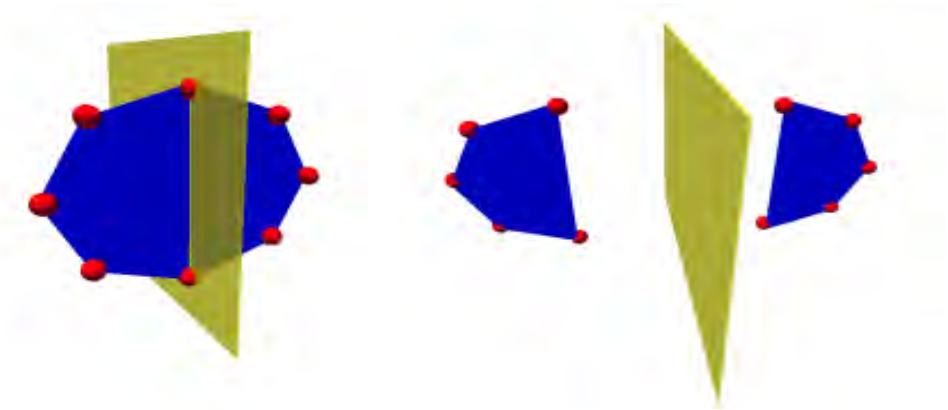


Figure 14.46 : On Plane vertices are added to both splits

We now correctly added the vertex to the front or back split but we cannot just progress to the next item in the list. If the vertex we just added and the next vertex in the list are on opposite sides of the plane we will need to treat these two vertices (the spanning edge) as a ray and intersect it with the plane.

The next section code does just that. If the next vertex in the list (CurrentVertex = i +1) is on the plane or is on the same side of the plane as the vertex we just added, then we can just continue to the next loop iteration where it will be added to its respective list.

```
        // If the next vertex is not causing us to span the plane then continue
        if (PointLocation[CurrentVertex]== CCollision::CLASSIFY_ONPLANE||
                                           PointLocation[CurrentVertex] ==
                                           PointLocation[i] )
            continue;
```

However, if we make it this far through the loop then the vertex we just added and the next vertex in the source polygon are on opposite sides of the plane and a new vertex will have to be added to both lists. This will be the vertex at the point of intersection between the edge and the plane.

We do this by making the vertex we just processed (*i*) the ray origin and then subtract the ray origin from the position of the next vertex in the list (CurrentVertex) which gives us our ray delta vector. We then use our CCollision::RayIntersectPlane function to calculate the *t* value of intersection.

```
            // Calculate the intersection point
            D3DXVECTOR3 vecOrigin  =(D3DXVECTOR3&)m_pVertex[i];
            D3DXVECTOR3 vecVelocity=(D3DXVECTOR3&)m_pVertex[CurrentVertex]
                                    - vecOrigin;
```

```
            CCollision::RayIntersectPlane( vecOrigin,
                                           vecVelocity,
                                           vecPlaneNormal,
                                           vecPlanePoint,
                                           fDelta,
                                           true );

            // create new vertex position
            CVertex NewVert;
            (D3DXVECTOR3&)NewVert = vecOrigin + (vecVelocity * fDelta);
```

When this function returns, local variable fDelta will contain the intersection point parametrically. This will be a value between 0.0 and 1.0 describing the position of intersection between the first and second vertex. You can see at the bottom of the above code that by scaling the ray delta vector (vecVelocity) by the *t* value of intersection (fDelta) and adding the result to the ray origin we have the position of the new vertex that we need to add.

Of course, we do not just need to calculate the new position of the vertex; we also have to interpolate the values of any other components that might be stored in the vertices, such as color and/or UV coordinates.

Figure 14.47 shows how a triangle polygon might be mapped to a given texture. Note the texture coordinates for v1 and v2. The red line shows where an intersection with the plane has occurred with this polygon and this is the point where a new texture coordinate needs to be created (percent = 0.5).

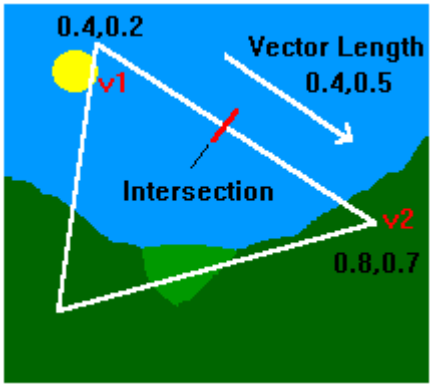


Figure 14.47

First we will subtract the first vertex texture coordinates from the second and end up with the vector length of the line between v1 and v2 in texture space. You can see that <0.8,0.7> - <0.4,0.2> = <0.4,0.5>. This is the direction and the length between texture coordinates v1 and v2. The great thing is that our RayIntersectPlane function returned a *t* value from the start of the line to the point where the intersection occurred. We can now use this value for texture coordinate interpolation, color interpolation, and even normal interpolation to generate all the other data we need for the newly created vertex. For example, imagine that the fDelta values returned from RayIntersectPlane was 0.5. This tells us that the edge intersects the plane exactly halfway between vertex v1 and vertex v2. If we multiply the texture coordinate delta vector by the *t* value (0.5 in this example) we get a vector of

<0.4*0.5, 0.5*0.5> = <0.2, 0.25>

Now we just have to add this scaled texture coordinate delta vector to the texture coordinates of the first vertex in the edge (v1) and we have the new texture coordinate of the vertex inserted at the intersection point.

New Texture Coordinates = <0.4, 0.2> + <0.2, 0.25> = <0.6, 0.45>

Incidentally, we use this same linear interpolation to generate a normal for the newly created vertex. Below we see the next section of the code that generates the normal and texture coordinate information for the new vertex.

```
            // Interpolate Texture Coordinates
            D3DXVECTOR3 Delta;
            Delta.x    = m_pVertex[CurrentVertex].tu - m_pVertex[i].tu;
            Delta.y    = m_pVertex[CurrentVertex].tv - m_pVertex[i].tv;
            NewVert.tu = m_pVertex[i].tu + ( Delta.x * fDelta );
            NewVert.tv = m_pVertex[i].tv + ( Delta.y * fDelta );

            // Interpolate normal
            Delta          = m_pVertex[CurrentVertex].Normal - m_pVertex[i].Normal;
            NewVert.Normal = m_pVertex[i].Normal + (Delta * fDelta);
            D3DXVec3Normalize( &NewVert.Normal, &NewVert.Normal );

            // Store in both lists.
            if (BackList)  BackList[BackCounter++]   = NewVert;
            if (FrontList) FrontList[FrontCounter++] = NewVert;

        } // Next Vertex

    } // End if spanning
```

Notice how the newly created vertex is added to both vertex lists. And that is the end of the loop.

At this point we have built the vertex lists for both of the polygons if applicable. Now it is time to actually allocate the new CPolygon objects (again, where applicable). In this first section we allocate the new polygon that will contain the front split vertex list. Obviously we only do this if the caller passed a valid FrontSplit pointer as a function parameter and if we have more than 2 vertices in the front list. Once the new polygon is allocated, we call its AddVertex method so that the CPolygon can reserve enough space in its vertex array for the correct number of vertices that we have compiled in the front vertex list. We then copy over the vertices that we have compiled in the temporary front list into the vertex array of CPolygon.

```
    // Allocate front face
    if (FrontCounter >= 3 && FrontSplit)
    {
        // Allocate a new polygon
        CPolygon * pPoly = new CPolygon;

        // Copy over the vertices into the new poly
        pPoly->AddVertex( FrontCounter );
        memcpy(pPoly->m_pVertex, FrontList, FrontCounter * sizeof(CVertex));

        // Copy over other details
        pPoly->m_nAttribID   = m_nAttribID;
        pPoly->m_vecNormal   = m_vecNormal;

        // Store the poly
        *FrontSplit = pPoly;

    } // End If
```

Notice in the above code that after we have copied the vertex data into the polygon, we also copy over the attribute ID and the normal of the source polygon. However much we split the polygon, every fragment will always exist on the same plane as the parent, so the normal can be safely inherited. The attribute id is also inherited into the child splits. This makes good sense because if we have a polygon with a wood texture applied and we split it, we get two polygons with that same wood texture applied. Finally, we assign the FrontSplit pointer passed by the caller to point to this newly created polygon so that when the function returns the caller will have access to it.

In the next section of code we do the same thing all over again for the back split polygon.

```
    // Allocate back face
    if (BackCounter >= 3 && BackSplit)
    {
        // Allocate a new polygon
        CPolygon * pPoly = new CPolygon;

        // Copy over the vertices into the new poly
        pPoly->AddVertex( BackCounter );
        memcpy(pPoly->m_pVertex, BackList, BackCounter * sizeof(CVertex));

        // Copy over other details
        pPoly->m_nAttribID   = m_nAttribID;
        pPoly->m_vecNormal   = m_vecNormal;

        // Store the poly
        *BackSplit = pPoly;

    } // End If
```

At this point our job is done so we release the temporary memory we used for the clipping/splitting process before returning true.

```
    // Clean up
    if (FrontList)      delete []FrontList;
    if (BackList)       delete []BackList;
    if (PointLocation)  delete []PointLocation;

    // Success!!
    return true;
}
```

Although that was a somewhat tricky function, the core process it performs is delightfully easy. It has been made much less reader friendly because it has many early out conditionals so that memory is not allocated unnecessarily, but nevertheless, it is still a pretty straightforward algorithm. You will see the Split method being used quite a lot during the construction of the clipped spatial tree.

We have now covered every thing we need to know about CPolygon; the structure used by our spatial trees to store static polygon data. We are now ready to look at ISpatialTree, the abstract base class for all of our new tree types and the interface our application will use to communicate with our trees.

14.11 The Abstract Base Classes

In the file ISpatialTree.h you will find the declaration of the ISpatialTree class. This is an abstract interface which exists to provide a consistent means of communication between our application and our various trees. As long as our quad-tree, oct-tree and kD-tree classes are derived from this interface and implement its methods, our application can use the same code to interface with any of these tree types. All the trees we have discussed share a lot of common functionality. For example, regardless of how they are built, they all contain an array of leaf nodes and an array of CPolygons assigned to each leaf node.

The ISpatialTree interface exposes all the methods an application will need to communicate with a spatial tree. We can think of it as defining a minimum set of functionality that must be implemented in our derived classes. It enforces for example that each of the trees we implement has an AddPolygon method so that an application can add CPolygons to its static geometry data. It enforces that our derived classes must implement an AddDetailArea method so that the application can register areas of interest for use during the build process. Another example of a method that must be implemented is the Build method which an application can use to instruct the spatial tree to compile its spatial information using the polygon data and detail areas that have been assigned to it. Further, in the ISpatialTree.h file, you will also see several structures and typedefs defined that will be used by all of our derived classes. Let us start to have a look at what is inside this file now.

The first structure defined in this file is the one that is used to pass detail information to and from the tree. This structure is very simple and has three members. The first two should contain the minimum and maximum extents of its world space AABB and the second is a context pointer that can be used by the application to point at any data it so chooses. Beyond the examples discussed earlier, this context pointer might point to some structure that contains settings to configure fog on the device. The detail area in this instance would be used to represent an area within the scene where fog should be enabled during rendering. The TreeDetailArea structure is shown below and is something we will see being used in a moment.

Excerpt From ISpatialTree.h

```
typedef struct _TreeDetailArea
{
    D3DXVECTOR3     BoundsMin;      // Minimum AABB extents of the area
    D3DXVECTOR3     BoundsMax;      // Maximum AABB extents of the area
    void          * pContext;       // A context pointer that can be assigned.

} TreeDetailArea;
```

Because all our various tree types will have support for the registration of detail areas, we have defined this structure in the main header file and have made the AddDetailArea and GetDetailAreaList methods members of the base class.

14.11.1 ILeaf – The Base Class for all Leaf Objects

Another thing that all of our trees will have in common is that they will all need the ability to store leaf information at terminal nodes in the tree. There are certain methods that we will want a leaf object to expose so that the tree (or application) can interface with a leaf more easily. For example, we will usually want a leaf to have an IsVisible method so that the application can query whether a given leaf is visible and should have its contents rendered. We will also want the ability to get back a list of all the detail areas and polygons that are assigned to a leaf. Furthermore, we know that each leaf will also need to store an AABB describing the region of space it occupies. Obviously, the leaf is only visible if this bounding box is contained either fully or partially inside the camera frustum.

Although we may want a leaf to contain much more information than this, the abstract ILeaf class is shown below and defines the base interface that our application will use to access and query a leaf. As with the ISpatialTree class, it is abstract, so it can never be instantiated directly. It must be derived from so that the function bodies can be implemented. These base interfaces force us to support and implement all the function in the base class when writing any tree type now or in the future. If we do not, we will get a compiler error. As long as we implement the functions specified in the base classes our application will happily work with the tree as its spatial manager with no changes to the code. This is because our application will only ever work with the ISpatialTree and ILeaf. As long as we derive our classes from these interfaces and implement the specified methods, we are in good shape to plug in whatever tree types we like down the road.

Below we see the class declaration for ILeaf.

```
class ILeaf
{
public:
    // Constructors & Destructors for This Class.
    virtual ~ILeaf() {};  // No implementation, but forces all derived classes
                             to have virtual destructors


    // Public Pure Virtual Functions for This Class.
    virtual bool            IsVisible           ( ) const = 0;
    virtual unsigned long   GetPolygonCount     ( ) const = 0;
    virtual CPolygon *      GetPolygon          ( unsigned long nIndex ) = 0;
    virtual unsigned long   GetDetailAreaCount  ( ) const = 0;
    virtual TreeDetailArea *GetDetailArea       ( unsigned long nIndex ) = 0;
    virtual void            GetBoundingBox      ( D3DXVECTOR3 & Min,
                                                  D3DXVECTOR3 & Max ) const = 0;
};
```

All of the methods shown above must be implemented in any derived leaf objects we create since they will be used by the spatial tree. Let us first discuss what these methods are supposed to be used for and what information they should return.

virtual bool IsVisible ( ) const = 0;
It is very important that a leaf expose this method as it is the application's means of testing whether a given leaf was found to be inside the frustum during the last update. This is very useful for example when performing queries on the tree such as testing to see if a dynamic object should be rendered.

virtual unsigned long GetPolygonCount ( ) const = 0;
This method returns the number of polygons contained in the leaf. This is very important and will be used often by our collision system. For example, whenever the player's position is updated, an AABB will be constructed and fed into the spatial tree's CollectLeavesAABB method. This method will traverse the tree and compile a list of all the leaves the passed AABB was found to intersect. In the case our broad phase collision pass, this will allow us to get back a list of leaf pointers after having passed the swept sphere's bounding box down the tree. We know that only the polygons in those leaves need to tested in the narrow phase. Of course, in order to test each polygon in the leaf we must set up a loop to count through and retrieve these polygons from the leaf. This method would be used to retrieve the number of polygons in the leaf that will need to be extracted and passed to the narrow phase.

virtual CPolygon * GetPolygon (unsigned long nIndex) = 0;
This method is used by the collision system (for example) to retrieve the CPolygon object in the leaf's polygon array at the index specified by the parameter. For example, if the GetPolygonCount function returns 15, this means that you will have to call GetPolygon 15 times (once for each index between 0 and 14) to extract each polygon pointer.

virtual unsigned long GetDetailAreaCount ( ) const = 0;
Each leaf may also contain an array of detail areas. That is, when the tree is built, any detail areas that had been registered with the tree prior to the build process commencing will have their pointers copied into a leaf's DetailArea array for any leaf whose bounding box intersects the bounding box of the detail area. A detail area will typically span many leaves, so it is quite typical during the build process for the same detail area to have its pointer stored in the DetailArea array of multiple leaf nodes.

virtual TreeDetailArea *GetDetailArea (unsigned long nIndex) = 0;
This method allows the application to fetch a detail area from the leaf. The input index must be between 0 and one less than the count returned from GetDetailAreaCount method.

virtual void GetBoundingBox(D3DXVECTOR3 & Min, D3DXVECTOR3 & Max) const = 0;
It can prove handy to be able to retrieve the AABB of a leaf, so this function provides that service. When passed two vectors, it will fill them with the minimum and maximum extents of the leaf's AABB.

14.11.2 ISpatialTree – The Base Tree Class

The ISpatialTree class is the abstract base class from which our spatial trees will be derived. It specifies the core functions that our derived classes must implement in order to facilitate communication with the application. Provided our derived classes (quad-tree, kD-tree, oct-tree, etc.) are all derived from this class and implement all the pure virtual functions specified, our application can work with all our tree objects using this common interface. This means that our application code will never change even if we decide to switch from a quad-tree to an oct-tree. The class also defines some types that the derived classes can use to declare polygon lists, leaf lists, and detail area lists. ISpatialTree contains no member variables and no default functionality, so we can never instantiate an object of this type. However, it does provide a consistent API that our derived tree classes must support in order for our application to effortlessly switch between tree types and build and query the given tree. How these functions are actually implemented in the derived classes is of no concern to the application, just so long as they perform the desired task.

ISpatialTree is contained in the file ISpatialTree.h, along with ILeaf, and its declaration is shown below.

```
class ISpatialTree
{
public:

    // Typedefs, Structures and Enumerators.
    typedef std::list<ILeaf*>           LeafList;
    typedef std::list<CPolygon*>        PolygonList;
    typedef std::list<TreeDetailArea*>  DetailAreaList;

    // Constructors & Destructors for This Class.
    virtual ~ISpatialTree() {};     // No implementation, but forces all derived
                                       classes to have virtual destructors

    // Public Pure Virtual Functions for This Class.
    virtual bool            AddPolygon       ( CPolygon * pPolygon ) = 0;
    virtual bool            AddDetailArea    (const TreeDetailArea & DetailArea)=0;
    virtual bool            Build               ( bool bAllowSplits = true ) = 0;
    virtual void            ProcessVisibility   ( CCamera & Camera ) = 0;
    virtual PolygonList    &GetPolygonList      ( ) = 0;
    virtual DetailAreaList &GetDetailAreaList   ( ) = 0;
    virtual LeafList       &GetLeafList         ( ) = 0;
    virtual bool            GetSceneBounds      ( D3DXVECTOR3 & Min,
                                                  D3DXVECTOR3 & Max ) = 0;
    virtual bool            CollectLeavesAABB   ( LeafList & List,
                                                  const D3DXVECTOR3 & Min,
                                                  const D3DXVECTOR3 & Max ) = 0;
    virtual bool            CollectLeavesRay    ( LeafList & List,
                                                  const D3DXVECTOR3 &RayOrigin,
                                                  const D3DXVECTOR3 &Velocity)= 0;

    // Public Optional Virtual Functions for This Class.
    virtual bool            Repair              ( ) { return true; }
    virtual void            DebugDraw           ( CCamera & Camera ) {};
    virtual void            DrawSubset          ( unsigned long nAttribID ) {};
};
```

The class contains a lot of function declarations and some typedefs which define how the system should work. Before moving on and examining how these methods are implemented in the derived class, let us first discuss what each method is expected to do so that we understand what is required of us in our derived class implementations.

typedef std::list<ILeaf*> LeafList
typedef std::list<CPolygon*> PolygonList
typedef std::list<TreeDetailArea*> DetailAreaList

These three type definitions are conveniences that the derived class can use to allocate variables of a certain type. For example, all of our trees will maintain a list of CPolygon pointers. Although you can feel free to store these polygon pointers in any format you choose in your derived class, the base class defines some handy type definitions allowing us to easily declare an STL list of polygons using the variable type 'PolygonList'. You can also see that there are type definitions that describe variables of type LeafList and DetailAreaList to be STL lists of those respective structures. Remember, these are just type definitions, so this class has no members and you can feel free not to use these types to store your polygon, leaf, and detail area data. However, we use variables of the types shown above in our derived classes to store the three key data elements managed by the tree (polygons, leaves, and detail areas).

virtual bool AddPolygon (CPolygon * pPolygon)
virtual bool AddDetailArea (const TreeDetailArea & DetailArea)

It makes sense that all of our derived tree classes will need to implement a function that will allow the application to add polygon and detail area data to its internal lists prior to its Build function being called. The two functions shown above must be implemented in the derived class so that the application has a means to do that. For example, when the application loads a static polygon from an IWF file, it will store it in a CPolygon structure and call the tree's AddPolygon method to register that polygon with the tree. This method will be implemented such that it stores the passed polygon pointer in its internal polygon pointer list (a variable of type PolygonList). The AddDetailArea is expected to be implemented in the same way so that the application can register areas of interest with the tree class. They will be used later in the build process to include areas that contain no polygon data (for example) within the volume of space partitioned by the tree.

virtual bool Build (bool bAllowSplits = true)

Every tree class that we develop will certainly need to expose a build function that allows the application to instruct the object to build its spatial hierarchy once it has registered all the polygon data and detail area data with the object. The build function of the derived class will be implemented differently depending on the type of tree being built. When the build function returns control back to the application, the spatial hierarchy will have been constructed and it will be ready for collision querying and visibility testing.

All of our trees will also have the option during creation to build a clipped on non-clipped tree. Thus, conditional logic will need to be put in place to determine what action should be taken if a polygon spans the bounds of a leaf in which it is partially contained. The bAllowSplits parameter to this function is the application's means of letting the tree know which type of build strategy should be employed. If the parameter is set to true, a clipped tree will be built. Any polygons that span multiple leaf nodes will be clipped to the bounds of each leaf in which it is contained and every leaf will ultimately contain a list

of polygons that are contained totally within its bounds. If this parameter is set to false, then any polygon that spans multiple leaf nodes will have its pointer stored in each of those leaf nodes.

> **Note:** Remember that clipping the polygons to the leaf nodes increases the polygon count of the level, potentially by a considerable amount. This means more polygons need to be rendered and more draw primitive calls have to be made. In our test level, this was a major problem with the oct-tree, where the polygon count increased between 60% and 90% when clipping was being used (depending on leaf size).

virtual void ProcessVisibility (CCamera & Camera)

Our application will also expect every tree class to implement the ProcessVisibility method, which exists to perform a hierarchy traversal with the passed camera and set any leaves that exist inside or partially inside the frustum as visible. This function, along with much of the rendering system of our tree classes, will be discussed in detail in the next chapter since this chapter is going to focus on the core building code. For now, just know that this function will be called by the scene prior to the tree being rendered. When this function returns, each leaf in the tree will know whether it is visible or not. When the application then issues a call to the ISpatialTree::DrawSubset method, the tree will know that it only has to render polygons that are contained in leaves that currently have their visible status set to true. Obviously, this will allow us to reject a lot of the geometry from being rendered most of the time.

virtual PolygonList &GetPolygonList ( )
virtual DetailAreaList &GetDetailAreaList ( )
virtual LeafList &GetLeafList ( )

These three methods must be implemented to allow the application to retrieve the polygon list, the detail area list and the list of leaves being used by the tree. This might be useful is the application would like to save the compiled tree data to file or would like to perform some custom optimization on the data. The GetPolygonList method is essential, since after the build process has completed, the internal list of polygons used by the tree might be quite different from the list the application originally registered with the tree. If clipping is being used, many of the polygons from the original list added by the application will have been deleted and replaced by the fragments they were divided into. Therefore, these functions are usually called after the build process so that the application has some way to access the data stored in the tree. Of course, the leaf list is not even compiled until the tree is built, so it would serve no purpose to call this function prior to calling the Build function.

virtual bool GetSceneBounds (D3DXVECTOR3 & Min, D3DXVECTOR3 & Max)

This method allows the application to retrieve an axis aligned bounding box that bounds the entire area of space partitioned by the tree. This is essentially the bounding box of the root node of the tree. Of course, this is not necessarily an AABB bounding only the static polygon data, it will also account for detail areas. Our derived classes will implement this method by simply returning the bounding volume of the root node.

virtual bool CollectLeavesAABB (LeafList & List, const D3DXVECTOR3 & Min, const D3DXVECTOR3 & Max)

This key method is one that the collision system uses to run queries on the tree. When this function is called, the application will pass an axis aligned bounding box and an empty leaf list (an STL list of ILeaf structures). This function should be implemented such that it will traverse the tree from the root node

and collect the leaves whose volumes are intersected by the passed AABB, attaching them to the input list. We can think of two situations where this function will be useful immediately.

When implementing the broad phase of our collision system we will wish to pass only polygons in the immediate area of the swept sphere's AABB to the narrow phase. In order to do this we will want to pass the swept sphere's AABB into the tree and get back a list of leaves that intersect it. It is only the polygons in those leaves that will have to be further checked at a lower level. Any polygons not contained in those leaves will never be considered by the collision system. This function will speed up our collision system by an order of magnitude.

Another time when our application will need to use this method is when determining in which leaves a dynamic object is located. Whenever a dynamic object has its position updated, we can pass its AABB into the tree and get back a list of intersected leaves. These are the leaves the dynamic object is currently contained within. This leaf list will be maintained internally for the object, but we can retrieve it via an ID prior to executing any draw calls. Before the scene renders any geometry, it will call the ISpatialTree::ProcessVisibility method which will determine which leaves are visible and which leaves are not. Since we know what leaves a dynamic object resides in, we can quickly test whether any of them were visible and take appropriate measures.

This function will be implemented as a simple recursive procedure that traverses the tree from the root node and performs and AABB/AABB intersection test between the passed AABB and the bounding volume of the node currently being visited. Once again, this is very fast because as soon as we find a node whose volume does not intersect the query volume, we never have to bother stepping into its child nodes, thus rejecting huge portions of the tree from having to be tested.

This function will need to be implemented differently in each derived class because the layout of the node structure and the way the tree is traversed will differ between tree types. For example, in a quad-tree we will need to step into four children at each node, while the oct-tree would recur into eight children. Of course, the underlying algorithm will be the same: perform AABB/AABB tests at each node and determining whether traversing into the children is necessary. Whenever we traverse into a leaf, it means that leaf is contained inside the query volume and it is added to the passed leaf list so that the caller will have access to it. We will discuss how to perform AABB/AABB intersections tests in a moment when we add that functionality to our CCollision class.

virtual bool CollectLeavesRay (LeafList & List, const D3DXVECTOR3 &RayOrigin, const D3DXVECTOR3 &Velocity)

This function has an almost identical purpose to the previously described function, only this time the query object is a ray instead of an AABB. There will be several times when we may wish to determine which polygons in the scene a ray intersects, and this function will allow us to do this efficiently.

The function is passed the ray origin and delta vectors along with an empty leaf list. The function should be implemented such that it will traverse the tree and return a list of leaves whose bounding volumes were intersected by the ray. Once again, this will have to be implemented slightly differently in each derived class as the way in which the tree is traversed is dependant on the node type (i.e., different numbers of children).

The function will step through each node in the tree and perform a ray/AABB intersection test with the node's bounding volume. If the ray does not intersect a volume then its child nodes do not have to be tested. For any node whose bounding box intersects the ray, we step into its children and perform the same process. Whenever we reach a leaf node, it means the ray must be partially contained within that leaf and the leaf structure is added to the leaf list so it can be returned to the caller.

virtual bool Repair ( ) { return true; }

The repair function is not an abstract function as it does have an implementation in ISpatialTree that essentially does nothing. This means that we do not have to bother implementing this function in the derived classes. However, it does give us an opportunity to perform some optimizations on the tree polygon data after the tree has been built. We will implement this function in our derived classes to perform a weld operation on the polygon vertices allowing to get rid of any redundant vertices (i.e., vertices that share the same position in 3D space and have the same attributes). This allows us to cut down on the number of vertices used by our scene quite dramatically in certain cases. In the case of GILES™, every face will have its unique vertices, so a cube constructed from 6 faces will actually contain 6*4=24 vertices. If the attributes of each face are identical, then at each corner of the cube there will be three vertices sharing the same space that have the same attributes (1 for each face that forms that corner). By performing a weld operation, we can collapse these three vertices at each corner into a single vertex that each of the three faces will index.

Another task that we will perform inside our Repair function will be the repair of T-junctions. T-junctions will be explained later, but as mentioned earlier, they frequently occur when lots of clipping has been performed. Since they cause very unsightly visual artefacts, we will definitely wish to repair them if we have built a clipped tree and intend to use the tree for rendering. Even if a non-clipped tree is being used, it is not uncommon for the source geometry to contain T-junctions that the artist may have inadvertently created, so we can repair those too.

If you do decide to derive your own tree types from ISpatialTree, you do not have to implement this step. Everything will still work fine because the default implementation will do nothing and just return. This method will be called by the scene after the Build function has returned and the tree has been completely built.

virtual void DebugDraw (CCamera & Camera) {}

This function is another function that has a default implementation that does nothing, so you do not have to implement it in your derived classes. However, it is often very useful for the application to be able to get some debug information about how the tree was constructed. As you have no doubt become aware, finding potential problems in recursive code can be very difficult, so if your spatial tree is not behaving in quite the way it should, it is useful to have this function available so that some rendering can be done to help you visualize how space has been partitioned. It is also useful when you are trying to configure the settings for your scene, such as the minimum leaf size (i.e., the size at which we make a node a leaf regardless of how much data it contains). As the leaf size will be directly related to the scale of your scene, you will likely want a way to see how the space has been partitioned.

In each of our derived tree classes we implement this method so that we can render the bounding boxes around each leaf node. Figure 14.51 shows a screen shots from Lab Project 14.1 with debug drawing enabled for a kD-tree (available via a menu option). On the code side, all the CScene::Render method

does after rendering the tree polygons and any dynamic objects, is call the tree's DebugDraw method. This method traverses the tree searching for visible leaf nodes. Once a leaf node is found it fetches the bounding box of the leaf and draws it using a line list primitive type.

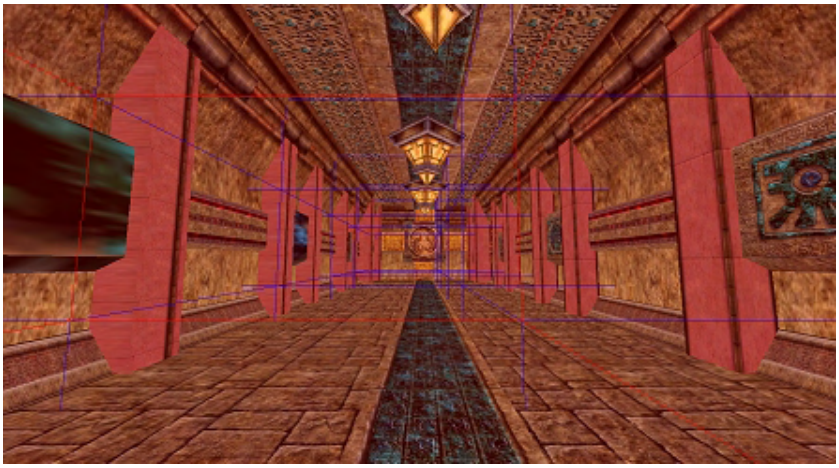


Figure 14.51

With debug draw enabled we can easily see the size of each leaf node and how the space is partitioned which, as mentioned, is very useful indeed when diagnosing various issues. In Figure 14.51 you can see that the bounding box of the leaf in which the camera is currently located is rendered in red instead of blue like all other leaf nodes. When the debug draw renders a red box it means that you are currently located in a leaf that contains something; either geometry or a detail area. However, if the leaf in which the camera is currently located is an empty leaf node the box is rendered in green as shown in Figure 14.52.

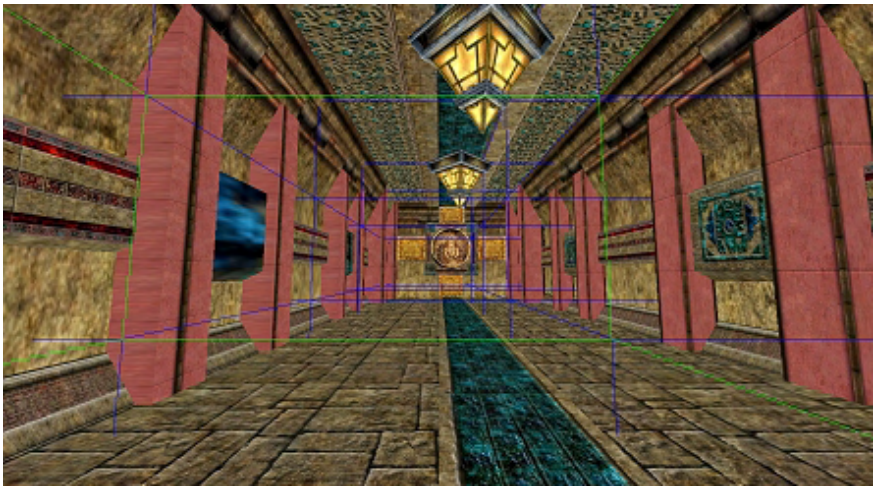


Figure 14.52

The DebugDraw method is a method that will have to be implemented slightly differently in each derived tree class. Although the output will be the same in all versions of this function (for the quad-tree, oct-tree and kD-tree) it is a function that needs to traverse the tree hierarchy and therefore, is dependant on the node structure of the tree being used and the number of children spawned from each node. However, each version of this function (for each tree type) will be almost identical. They will all traverse the tree looking for leaf nodes and then render bounding boxes using the bounding volume information. We will look at the code to these functions later in the lesson.

virtual void DrawSubset (unsigned long nAttribID) {}

In theory, rendering a spatial tree is very easy. For example, we could just traverse the hierarchy and collect the polygons from the visible leaf nodes and copy them into dynamic index and vertex buffers. Unfortunately, with such a scheme, memory copying really hampers performance when we are building those render buffers. Using a dynamic index buffer only approach is certainly much better, but still not optimal. Rendering a tree efficiently in a way that does not hamper the performance of powerful 3D

graphics cards is not so easy and in the next chapter we will discuss the rendering system that we have put in place that all of our spatial trees will use to collect and render their geometry.

> **Note:** Since the spatial tree rendering system will be rather involved, we will dedicate a good portion of the next chapter entirely to this one topic. Although this rendering system is used in Lab Project 14.1, note that this same lab project will be discussed over these two chapters. In this chapter we will discuss building the spatial tree and the upgrade of the collision system to use the spatial tree during its broad phase pass. In the next lesson we will discuss exactly how the tree data is converted into hardware friendly render batches so that we can minimize draw primitive calls and maximize batch rendering. When examining the code to Lab Project 14.1, it may be best for you to ignore how the tree is rendered for now.

The DrawSubset function will need to be implemented by each tree class so that the application can instruct the tree that it needs to render all visible polygons belonging to the passed subset. As we know, our scene is responsible for setting textures and materials and it does not want to set the same one more than once if it can be avoided. That is, we wish to setup a given attribute and then render all the polygons in the scene that use it. This minimizes render state changes and DrawPrimtive calls, which can really help performance. The CScene::Render method will first call the ISpatialTree::ProcessVisibility method prior to any rendering taking place. This instructs the tree to traverse its hierarchy and mark any leaves that are inside the view frustum as being visible. Our scene will then loop through each attribute used by the scene and call this method, each time passing in the current attribute/subset being processed. This will instruct the tree to render any polygon in any of the currently visible leaves that belong to this subset. As you will see in the next lesson, doing this efficiently means putting quite a complex system in place.

Although we have not yet discussed the implementation details of any of the functions declared in the base class, discussing ISpatialTree and the methods it exposes has given us an understanding of how our tree will work, the functions that we are expected to implement in our derived classes, and the way in which our application will communicate with our trees both when querying it and rendering it. With that said, let us start coding.

14.11.3 CBaseTree & CBaseLeaf – Base Functionality Classes

For the most part, in this course we have not really had to implement chains of derived classes. At most we have sometimes implemented a base class and a derived class as this makes the code clearer to read, easier to learn, and it makes program flow easier to follow (and of course, it is often good OOP). However, sometimes it makes sense to have an additional layer of inheritance when all your derived classes will share many properties and would need to have identical code duplicated for each. For example, regardless of the tree types we implement, they will all share a lot of common ideas. Each tree type will store a list of polygons, leaves, and detail areas, and will need to implement methods that allow the application to add and retrieve elements to/from these lists.

Likewise, the leaf structure used by each tree will essentially be the same and will have the same tasks to perform. Each will store a bounding box and will need to implement methods to add polygons and detail areas to its internal lists and expose functions which allow the tree to set its visible status during a

visibility update. Furthermore, although we will not discuss the rendering subsystem in this chapter, the same system will be used by each of the derived tree classes. If we were simply going to derive our quad-tree, kD-tree, and oct-tree classes directly from ISpatialTree, we would have to implement this code in each of our derived classes. This is wasteful since we would be duplicating identical code in each derived class. For example, the AddPolygon method will be implemented identically in each tree type; it will be a simple function that accepts a CPolygon pointer and stores that pointer in the tree's polygon list.

To minimize redundancy across these common tasks, we will include a middle layer called CBaseTree. CBaseTree will be derived from ISpatialTree and will implement many of the functions that are common to all tree types. It will also implement all the code to the rendering system which will be used by each derived tree class. Although we will never be able to instantiate an object of type CBaseTree (as it does not implement all the method from ISpatialTree) it will provide all of the housekeeping code. We will derive our quad-tree, oct-tree, and kD-tree classes from CBaseTree.

Due to the fact that CBaseTree contains a lot of code that is common to all tree types, implementing our actual tree classes will involve just the implementation of a few functions. For example, when implementing a quad-tree class, we essentially just have to implement the methods from ISpatialTree that are not implemented in CBaseTree. One such method is the Build method which will obviously be different for each tree type. The only other method we will have to implement in the lower level tree classes are the query routines such as CollectLeavesAABB, CollectLeavesRay, and ProcessVisibility. As these methods are tree traversal methods, they must be implemented by the actual types. Thus, if you open up the CQuadTree.cpp file for example, you will see that very little code is contained in there, as most of the common functionality is contained inside CBaseTree.

In this next section we will discuss the implementation of the CBaseTree methods that are used during the building phase. The rendering system contained in CBaseTree will be discussed in the following lesson, so its methods will be removed from the class declaration at this time. In this lesson, we are just concentrating on the CBaseTree methods used by the tree building process and the methods associated with querying the tree (such as the method used by our collision system). The code to CBaseTree and CBaseLeaf are stored in the files CBaseTree.h and CBaseTree.cpp.

14.11.4 CBaseLeaf – The Source Code

CBaseLeaf is derived from ILeaf and implements all the functionality of the base class. That is, all of our trees will store leaves of type CBaseLeaf. The class declaration is shown below and we will discuss it afterwards. Notice how the first pool of function declarations are those from ILeaf that will be implemented in this class. This is followed by some functions which CBaseTree will need to communicate with a leaf.

```
class CBaseLeaf : public ILeaf
{
public:

    // Constructors & Destructors for This Class.
    virtual ~CBaseLeaf( );
```

```
            CBaseLeaf( CBaseTree *pTree );

    // Public Virtual Functions for This Class (from ISpatialTree).
    virtual bool            IsVisible           ( ) const;
    virtual unsigned long   GetPolygonCount     ( ) const;
    virtual CPolygon *      GetPolygon          ( unsigned long nIndex );
    virtual unsigned long   GetDetailAreaCount  ( ) const;
    virtual TreeDetailArea* GetDetailArea       ( unsigned long nIndex );
    virtual void            GetBoundingBox      ( D3DXVECTOR3 & Min,
                                                  D3DXVECTOR3 & Max ) const;

    // Public Functions for This Class.
    void                    SetVisible          ( bool bVisible );
    void                    SetBoundingBox      ( const D3DXVECTOR3 & Min,
                                                  const D3DXVECTOR3 & Max );
    bool                    AddPolygon          ( CPolygon * pPolygon );
    bool                    AddDetailArea       ( TreeDetailArea * pDetailArea );

protected:

    // Protected Structures, Enumerators and typedefs for This Class.
    typedef std::vector<CPolygon*>          PolygonVector;
    typedef std::vector<TreeDetailArea*>    DetailAreaVector;

    // Protected Variables for This Class
    PolygonVector       m_Polygons;     // Array of polygon pointers in this leaf.
    DetailAreaVector    m_DetailAreas;  // Array of detail area pointers in leaf.
    bool                m_bVisible;     // Is this leaf visible or not?
    D3DXVECTOR3         m_vecBoundsMin; // Minimum bounding box extents
    D3DXVECTOR3         m_vecBoundsMax; // Maximum bounding box extents.
    CBaseTree          *m_pTree;        // The tree that owns this leaf
};
```

Looking at the above class declaration we can see that it implements those methods from the base class that allows the application to perform queries on the leaf such as retrieving the leaf's visible status or retrieving its bounding volume. Following these declarations are the functions that are new to this class which CBaseTree and any class derived from it can use to add detail areas or polygons to the leaf's arrays. It also exposes methods which CBaseTree (or any class derived from it) can use to set the visibility status of a leaf (inside the ProcessVisibility function) and methods allowing the tree building functions to set the leaf's AABB.

Following that are two type definitions called PolygonVector and DetailAreaVector. These are STL vectors which the leaf will use to store the polygon pointers and detail area pointers assigned to it. Unlike the CBaseTree which stores its polygon and detail area data in STL lists (linked lists) for efficient manipulation during the tree building process, the data that ends up being assigned to a leaf remains static once the leaf has been built. Therefore, we use vectors for faster access.

Finally, at the bottom of the declaration we can see the member variables that each leaf structure contains. They are discussed below, although their meaning will most likely be self-explanatory given their names.

PolygonVector m_Polygons
This member is an STL vector that will be used to store the polygons assigned to this leaf. During the build process, once a node has been reached which suits the criteria for being a terminal node (such as its bounding volume is sufficiently small or the number of polygons is below a certain threshold) a new CBaseLeaf object will be created and added to the tree's leaf list. The leaf structure will also be attached to the node in the tree and the polygon data that made it into that node will be added to the leaf via the CBaseLeaf::AddPolygon method. This method will store the passed CPolygon pointer in this vector.

DetailAreaVector m_DetailAreas
In an almost identical manner to the method described above, this vector will be used to store pointers to any detail objects that exist inside or partially inside the leaf.

bool m_bVisible
This member is used internally by the leaf to record its current visibility status. If, during the last ProcessVisibility test, its bounding box was found to be inside or partially inside the frustum, the tree would have set this boolean to true through the use of the CBaseLeaf::SetVisible method. The application (or the tree itself) can query the visibility status of a leaf using the CBaseLeaf::IsVisible method, which simply returns the value of this boolean.

D3DXVECTOR3 m_vecBoundsMin
D3DXVECTOR3 m_vecBoundsMax
Each leaf will store a bounding volume (an AABB) which will be set by the tree during the build process. When a leaf is created, the bounding box of the polygon data and the detail area data that made it into that node is computed. The tree will then use the CBaseLeaf::SetBoundingBox method to set the bounding box for the leaf.

CBaseTree * m_pTree
Each leaf will store a copy of the pointer to the tree which owns it. We will see how this pointer is used in the next chapter when we discuss rendering.

Let us now take a look at the method implementations shown above. Remember, if you are following along with the lab project source code files open, you will see many other method in CBaseTree that we have not discussed here. These are methods related to the rendering system which will be fully explained in the next lesson.

Constructor - CBaseLeaf

The CBaseLeaf constructor could not be simpler. We just initialize its visibility status to true. We will assume that the default state of any node/leaf in the tree is visible so that if for some reason we do not wish to perform the ProcessVisibility pass, all leaves will be rendered.

```
CBaseLeaf::CBaseLeaf( CBaseTree *pTree )
{
    // Reset required variables
    m_bVisible          = true;
```

```
     .. render data ..

    // Store the tree
    m_pTree = pTree
}
```

When the leaf is first created its polygon and detail area vectors will be empty and its bounding box will be uninitialized. Note that the leaf also accepts and stores a pointer to the base tree to which it belongs.

Setting/Getting the Leaf's Visibility Status - CBaseLeaf

The method that allows the application to query the visibility status of a leaf with a simple one line function that returns the value of the leaf's visibility flag.

```
bool CBaseLeaf::IsVisible( ) const
{
    return m_bVisible;
}
```

Likewise, the method that allows our tree classes to set the visibility status of a leaf (during the ProcessVisibility pass) is a one line function that sets the boolean member equal to the boolean parameter passed.

```
void CBaseLeaf::SetVisible( bool bVisible )
{
    // Flag this as visible
    m_bVisible = bVisible;
}
```

AddPolygon – CBaseLeaf

The AddPolygon method is called by the tree building process whenever a terminal node is encountered. A new CBaseLeaf object is allocated and its pointer is stored in the terminal node. The AddPolygon method will then be called for each polygon that made it into the terminal node. The method adds the passed CPolygon polygon pointer to the leaf's polygon vector (with exception handling to return false should an error occur in the process).

```
bool CBaseLeaf::AddPolygon( CPolygon * pPolygon )
{
    try
    {
        // Add to the polygon list
        m_Polygons.push_back( pPolygon );

    }
```

```
    catch ( ... )
    {
        return false;

    }

    // Success!
    return true;
}
```

AddDetailArea – CBaseLeaf

When a leaf is created and added to a terminal node during the building process, any detail area that made it into that node will also be added to the leaf. The AddDetailArea adds the passed TreeDetailArea pointer to the leaf's internal detail area vector. The function code is shown below.

```
bool CBaseLeaf::AddDetailArea( TreeDetailArea * pDetailArea )
{
    try
    {
        // Add to the detail area list
        m_DetailAreas.push_back( pDetailArea );

    } // End Try Block

    catch ( ... )
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

Retrieving the Polygon Data from a Leaf – CBaseLeaf

It will often be necessary for the application to retrieve the polygon data stored in a leaf. This is certainly true in our broad phase collision step when we will send the swept sphere's AABB down the tree and get back a list of intersecting leaves using the tree's CollectLeavesAABB method. Once this list of leaves is returned, the broad phase can fetch each polygon from each returned leaf and test it more thoroughly (first with an AABB/AABB test between the AABB of the swept sphere and the AABB of the polygon and then with the more expensive narrow phase if the prior test returns true for an intersection).

For the collision system to get this information, it must know how many polygons are stored in a leaf and have a way to access each polygon in that leaf. Below we see the implementations of the

CBaseLeaf::GetPolygonCount and CBaseLeaf::GetPolygon methods which are part of the ISpatialTree interface. The GetPolygonCount method simply returns the size of the leaf's internal CPolygon vector.

```
unsigned long CBaseLeaf::GetPolygonCount( ) const
{
    // Return number of polygons stored in our internal vector
    return m_Polygons.size();
}
```

The GetPolygon method accepts an index (in the range of zero to the value returned from GetPolygonCount - 1) and returns the CPolygon pointer stored at that location in the vector.

```
CPolygon * CBaseLeaf::GetPolygon( unsigned long nIndex )
{
    // Validate the index
    if ( nIndex >= m_Polygons.size() ) return NULL;

    // Return the actual pointer
    return m_Polygons[nIndex];
}
```

Retrieving the Detail Area Data from a Leaf – CBaseLeaf

An application may also wish to retrieve the information about which detail areas are stored in a leaf. For example, perhaps you have inserted a specific detail area that has a context pointer that points to a structure filled with fog settings. Whenever the camera is in a leaf which contains such a detail area, the pipeline's fog parameters could be set and enabled as described by the detail area's context pointer. Such a task could be performed by finding the leaf the camera is currently in and then fetching the number of detail area areas assigned to this leaf. You could then set up a loop to extract and test each detail area. If a detail area is found which has a context pointer that points to a fog structure, fog could be enabled.

In order to do this we would need to be able to fetch the number of detail areas in a leaf and expose a means for those detail areas to be retrieved and examined. The CBaseLeaf::GetDetailAreaCount is a simple function that returns the size of the leaf's detail area vector:

```
unsigned long CBaseLeaf::GetDetailAreaCount( ) const
{
    // Return number of polygons stored in our internal vector
    return m_DetailAreas.size();
}
```

The CBaseLeaf::GetDetailArea method is passed an index between 0 and the value returned by the GetDetailAreaCount function minus 1. It returns the TreeDetailArea pointer stored at that position in the vector.

```
TreeDetailArea * CBaseLeaf::GetDetailArea( unsigned long nIndex )
{
    // Validate the index
```

```
    if ( nIndex >= m_DetailAreas.size() ) return NULL;

    // Return the actual pointer
    return m_DetailAreas[nIndex];
}
```

Setting and Retrieving a Leaf's Bounding Box - CBaseLeaf

The application may want to retrieve the leaf bounding box for custom intersection routines. The CBaseLeaf::GetBoundingBox method accepts two 3D vector references and populates them with the minimum and maximum extents of the leaf's AABB.

```
void CBaseLeaf::GetBoundingBox( D3DXVECTOR3 & Min, D3DXVECTOR3 & Max ) const
{
    // Retrieve the bounding boxes
    Min = m_vecBoundsMin;
    Max = m_vecBoundsMax;
}
```

The application will never need to set the bounding box of a leaf since that is the responsibility of the tree building process. Therefore, the SetBoundingBox method is not a member of ILeaf (the API used by the application); it is added to CBaseLeaf instead. The derived tree classes will call this method when the leaf is created at a terminal node and pass it the minimum and maximum extents of an AABB which represents the area of the terminal node (the leaf). Below we see the code to the function that will be called by the tree building process after a leaf has been allocated at a terminal node.

```
void CBaseLeaf::SetBoundingBox( const D3DXVECTOR3 & Min, const D3DXVECTOR3 & Max )
{
    // Store the bounding boxes
    m_vecBoundsMin = Min;
    m_vecBoundsMax = Max;
}
```

We have now covered all of the housekeeping methods in our CBaseLeaf class (the class that will be used to represent polygon and detail area data at terminal nodes in the tree). This object will be used by all of our derived tree classes to represent leaf data.

There are some additional members and methods that are not shown here as they relate to the rendering subsystem of CBaseTree. These will be discussed in the following lesson. For now, we have covered all the important code that will be needed to understand the construction and configuration of leaf data both during the build process and during collision queries.

14.11.5 CBaseTree – The Source Code

CBaseTree is the class that our other tree classes will be derived from. It implements many of the methods required by the ISpatialTree interface and as such, by deriving our tree classes from this class,

we avoid having to duplicate such housekeeping functionality. As discussed earlier, with CBaseTree in place, we can easily create almost any tree type we want simply by deriving a new class from it and implementing its build and query traversal functions. CBaseTree will be responsible for managing the polygon, leaf, and detail area lists and implementing the methods from ISpatialTree that allow the application to register data with the tree prior to the build process.

CBaseTree will also implement other methods that our derived classes will not want to worry about, such as the repair method from ISpatialTree. The CBaseTree::Repair method will repair T-junctions in the geometry and can be called after the tree has been built. Although we will not discuss the rendering logic in this chapter, this class also implements the DrawSubset method responsible for rendering the various subsets of static polygons stored within the tree. It also has an implementation of a ProcessVisibility method, although this method must be overridden in your derived class. The derived version must call the CBaseTree version prior to performing its visibility traversal. Although the meaning of the CBaseTree version of this method will not be fully understood until the next lesson, this method essentially gives CBaseTree a chance to flush its render buffers before they get refilled by the derived class. This will be explained later, so do not worry too much about it for now.

Finally, CBaseTree implements some utility methods that can be called by the derived class to make life easier. An example of this happens when implementing the DebugDraw method in your derived classes. As discussed earlier, the DebugDraw method traverses the hierarchy and renders a bounding box for any visible leaf node. This allows us to see each leaf's volume when running our application. However, although this function essentially has the same task to perform for each tree type, the method must be implemented for each tree type due to the fact that we traverse an oct-tree differently from how we traverse a quad-tree. But the traversal code is very small; it is really the rendering of the bounding box which takes a bit more code and will be identical for each tree type.

Because of this fact, CBaseTree will expose a method called DrawBoundingBox which can be called from a derived class and passed an AABB. This method will take case of the actual construction and rendering of the bounding box to the frame buffer. Therefore, all we have to do when we implement the DebugDraw method in our derived tree classes is write code in there that traverses the tree looking for visible leaf nodes. As we find each one, we just call the CBaseTree::DrawBoundingBox method and pass it the AABB of the leaf in question. This function will then render the bounding box and we minimize redundant code. This class also implements another DebugDraw helper function called CBaseTree::ScreenTint that can be passed a color and will alpha blend a quad over the entire frame buffer. We use this in our derived class's DebugDraw method to tint the screen red when the camera enters a leaf that has data contained in it (a non-empty leaf).

Below we see the class declaration of CBaseTree. We have removed any functions, structures, and member variables related to its rendering subsystem since we will introduce these in the following lesson.

```
class CBaseTree : public ISpatialTree
{
public:

    // Friend list for CBaseTree
    friend void CBaseLeaf::SetVisible( bool bVisible );
```

```
    // Constructors & Destructors for This Class.
    virtual ~CBaseTree( );it easily
    CBaseTree( LPDIRECT3DDEVICE9 pDevice, bool bHardwareTnL );

    // Public Virtual Functions for This Class (from base).
    virtual bool            AddPolygon       ( CPolygon * pPolygon );
    virtual bool            AddDetailArea    ( const TreeDetailArea& DetailArea );
    virtual bool            Repair             ( );
    virtual PolygonList    &GetPolygonList     ( );
    virtual DetailAreaList &GetDetailAreaList  ( );
    virtual LeafList       &GetLeafList        ( );
    virtual void            DrawSubset         ( unsigned long nAttribID );
    virtual void            ProcessVisibility  ( CCamera & Camera );

protected:
    bool          PostBuild         ( );
     void         DrawBoundingBox   ( const D3DXVECTOR3 & Min,
                                      const D3DXVECTOR3 & Max,
                                      ULONG Color, bool bZEnable = false );

    void          DrawScreenTint    ( ULONG Color );
    void          CalculatePolyBounds ( );
    void          RepairTJunctions  ( CPolygon * pPoly1,
                                      CPolygon * pPoly2 );

    // Protected Variables for This Class.
    LPDIRECT3DDEVICE9          m_pD3DDevice;
    bool                       m_bHardwareTnL;
    LeafList                   m_Leaves;
    PolygonList                m_Polygons;
    DetailAreaList             m_DetailAreas;
};
```

A few things in the above declaration are worthy of node. First, notice how the CBaseLeaf::SetVisible method is made a friend, so that we can access it from within this class. The tree will need to be able to set the visible status of a leaf when it is performing its visibility pass. Also notice that it has a method called PostBuild. This method will be called by the derived class after the tree has been constructed. For example, our CQuadTree class will call this method at the very end of its Build function after the tree has been constructed. The PostBuild method does two things. The first thing it does is call the CBaseTree::CalculatePolyBounds method which will loop through each CPolygon stored in the tree, calculate its bounding box, and store that bounding box in the polygon structure. The polygon bounding boxes will be used by our broad phase collision step so that only polygons whose bounding boxes intersect the AABB of the swept sphere get passed onto the narrow phase. This is another example of a function that would otherwise need to be implemented in each of the derived classes if it were not for CBaseTree managing such functionality. The second thing the CBaseTree::PostBuild method does is call CBaseTree::BuildRenderData. This method is not shown above because we will discuss it in the next chapter (basically, it configures the rendering subsystem for CBaseTree).

We can see in the above declaration that the CBaseTree also has a method called RepairTJunctions. It is called from the CBaseTree::Repair method to mend any T-junctions which were introduced during the building phase. If the tree is not being used for rendering and you wish to repair T-junctions introduced in the build phase, then the application should call the Repair method after the tree has been built. If the tree is being used for rendering then there is no need, because the BuildRenderData method will

automatically call this method before preparing the render data. Keep in mind that if a tree is being used for rendering, we definitely want to always repair T-junctions to remove unsightly artifacts. We will examine what T-junctions are and how they can be repaired later in this lesson.

Let us now discuss the member variables declared in CBaseTree which will be used for storage by the derived classes.

LPDIRECT3DDEVICE9 m_pD3DDevice
If the application wishes to use the tree for rendering, it should pass a valid pointer to a Direct3D device into the constructor of the derived class, which will pass it along to the base class for storage in this member. If NULL is passed to the derived class constructor then NULL will be passed to the constructor of CBaseTree as well and this parameter will be set to NULL. If this parameter is NULL, no render data will be built when PostBuild is called at the end of the derived class's Build method.

bool m_bHardwareTnL
This boolean will also be set via the application passing its value to the derived class constructor, which in turn will be passed to the CBaseTree constructor and stored in this member. We have used a boolean value like this many times before to communicate to a component whether the device being used is a hardware or software vertex processing device. We will see in the following chapter how the base tree class will need to know this information when building the vertex and index buffers for its render data.

LeafList m_Leaves
PolygonList m_Polygons
DetailAreaList m_DetailAreas
Earlier we saw that the types LeafList, PolygonList, and DetailAreaList were defined in ISpatialTree as STL lists (linked lists) of ILeaf, CPolygon, and TreeDetailArea objects, respectively. These linked lists store the leaves, polygons, and detail areas when they are added to the tree. At any point (even pre-build) the m_Polygons and m_DetailAreas lists will contain all the polygons and details areas registered and in use by the tree at that time. m_Leaves will only contain valid data after the tree has been built and the leaves have been created.

Before the tree is built, the application will need a way to register detail areas and polygons with the tree so that they can be used in the building process. For example, every time we load a polygon from an IWF file we will call the AddPolygon method that will add it to the above list. After all the data has been added to the tree, but prior to the Build function being called, this list will contain all the polygon data that will be partitioned by the build process. After the build process has completed however, the polygons stored in the m_Polygons array (and in the leaf polygon arrays) may be different from the original list that existed prior to tree compilation. This is certainly true if a clipped tree is being constructed since many of the polygons in the original list will be deleted and replaced by two split fragments. Likewise, even if the lists are the same post-build, if you call the Repair function to mend T-junctions, additional triangles will be inserted to repair those T-junctions. The important point is that whether post-build or pre-build, the m_Polygons and m_DetailAreas lists will always contain all the polygons and detail areas being used by the tree. In the post-build case, we will also have pointers to all the polygons and detail areas in these lists stored in the polygon and detail area vectors of the leaves as well.

Constructor – CBaseTree

The CBaseTree constructor is called from the constructor of the derived class and is passed two parameters supplied by the application. If pDevice is not NULL then it means that the application wishes to use this tree for rendering and the rendering subsystem will be invoked (described in the next chapter). The device pointer and the boolean describing the hardware/software status of the device are stored in the base class variables and the reference count of the device is incremented.

```
CBaseTree::CBaseTree( LPDIRECT3DDEVICE9 pDevice, bool bHardwareTnL )
{
    // Store the D3D Device and other details
    m_pD3DDevice    = pDevice;
    m_bHardwareTnL  = bHardwareTnL;

    // Add ref the device (if available)
    if ( pDevice ) pDevice->AddRef();
}
```

There are a few additional lines to this constructor not shown here since they concern the rendering system which we will introduce later.

AddPolygon - CBaseTree

All of our derived classes will have a few things in common. One is the need for the application to add polygon data to its internal arrays in preparation for the build process. The AddPolygon method will be called by the application every time it loads a static polygon which it would like to be part of the data set that is to be spatially partitioned. If you look in the CScene::ProcessVertices function of Lab Project 14.1, you will see that a new line has been inserted that adds the vertices of the polygon currently being processed to the spatial tree being used by the scene. This is the function it calls, and as you can see, it simply adds the passed polygon pointer to its internal polygon list.

```
bool CBaseTree::AddPolygon( CPolygon * pPolygon )
{
    try
    {
        // Add to the polygon list
        m_Polygons.push_back( pPolygon );

    } // End Try Block
    catch ( ... )
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

Only after the application has called the AddPolygon method for every static polygon it wishes to be stored in the tree should it call the tree's Build method. We will see later how the derived class's Build function will use the polygon list to build its spatial hierarchy.

AddDetailArea - CBaseTree

The application will also need the ability to register detail areas with the tree prior to the Build method being executed. This method is passed a TreeDetailArea structure which will be added to the tree's detail area list.

```
bool CBaseTree::AddDetailArea( const TreeDetailArea & DetailArea )
{
    try
    {
        // Allocate a new detail area structure
        TreeDetailArea * pDetailArea = new TreeDetailArea;
        if ( !pDetailArea ) throw std::bad_alloc();

        // Copy over the data from that specified
        *pDetailArea = DetailArea;

        // Add to the area list
        m_DetailAreas.push_back( pDetailArea );

    } // End Try Block

    catch ( ... )
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

This method should be called by the application for each detail area (AABB) it would like to register with the tree prior to tree compilation. These detail areas can be used to force the tree to partition space that contains no static polygon data or not partition space that contains much polygon data. We will understand exactly how this is achieved later when we look at the Build methods for the various derived classes.

Retrieving Data from the Tree - CBaseTree

There may be times after the build process has been completed (or pre-build with respect to the polygon and detail area lists) when an application would like to retrieve a list of all the polygons, leaves, and

detail areas being used by the entire tree. The following three methods are simple access functions which return the leaf list, polygon list, and the detail area list back to the caller.

```
CBaseTree::LeafList & CBaseTree::GetLeafList()
{
    return m_Leaves;
}
```

```
CBaseTree::PolygonList & CBaseTree::GetPolygonList()
{
    return m_Polygons;
}
```

```
CBaseTree::DetailAreaList & CBaseTree::GetDetailAreaList()
{
    return m_DetailAreas;
}
```

AddLeaf - CBaseTree

Our derived classes will need a way to add leaves to their arrays during the build process. When a terminal node is encountered and a new CBaseLeaf is allocated and attached to the terminal node, we will also want to store that leaf's pointer in the tree's leaf list. Since this functionality will be the same for each tree type we will implement, this method in CBaseTree will save us the trouble of reinventing the wheel in each of our derived classes. The code simply adds the passed leaf pointer to the tree's internal leaf list.

```
bool CBaseTree::AddLeaf( CBaseLeaf * pLeaf )
{
    try
    {
        // Add to the leaf list
        m_Leaves.push_back( pLeaf );

    } // End Try Block

    catch ( ... )
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

PostBuild - CBaseTree

The PostBuild method of CBaseTree should be called by the derived class after the tree has been constructed. In our derived classes, we will call this function at the very bottom of their Build method. PostBuild first calls the CBaseTree::CalculatePolyBounds method (which we will discuss next) which loops through each polygon in the tree's polygon list and calculates and stores its AABB. It then calls the CBaseTree::BuildRenderData method which allows the CBaseTree to initialize the render system with the tree data that has just been constructed. Many of you used to working with MFC might recognize this type of relationship, as it is much like creating a window. In MFC, methods can be overridden that allow your application to perform some default processing either just before or just after the window has been created. In this case, we are letting the base tree know that the tree building process is complete so that it can be prepared for rendering. The code to the function is shown below.

```
bool CBaseTree::PostBuild( )
{
    // Calculate the polygon bounding boxes
    CalculatePolyBounds( );

    // Build the render data
    return BuildRenderData( );
}
```

The CalculatePolyBounds method will be discussed next but we will defer our discussion of the BuildRenderData method until the next lesson where we discuss the rendering system. If the application did not pass a 3D device pointer into the derived tree's class constructor, this function will simply return without doing anything. In this case, the tree's rendering functionality will not be available but it can still be queried by the collision system.

CalculatePolyBounds - CBaseTree

The CalculatePolyBounds function will be used by all of our derived tree types. It will particularly important for an efficient broad phase collision step. Rather than ask the tree to return a list of leaves that intersect the swept sphere's AABB and the send the polygons in those leaves immediately on to the narrow phase, we will introduce an intermediate broad phase step which will be cheap but very effective. After we have retrieved the leaves that the swept sphere intersects, we will test each polygon in those leaves against the swept sphere AABB using their respective bounding boxes. Only polygons whose bounding boxes intersect the bounding box of the swept sphere will need to be passed to the more expensive narrow phase. This is an inexpensive test that gives us a very impressive performance enhancement.

> **Note:** Performing the polygon bounding box test in our lab testing really did increase speed by an impressive amount. For example, on one of the levels we were testing with the collision system, even when using the tree to collect only the relevant leaves, but without performing the polygon bounding box tests, our frame rates fell to ~30fps when querying a particularly dense leaf (i.e., a leaf with many polygons in it). After adding the polygon bounding box test, many polygons were rejected and did not get sent to the narrow phase and our frame rate in that same leaf increased to a solid 560fps. That is

obviously quite a savings and well worth the small amount of additional memory the bounding boxes add to each polygon.

This method is called from the CBaseTree::PostBuild method, which itself is called from the Build function of the derived class after the tree has been fully constructed. The job of the function is straightforward enough; loop through each static polygon stored in the tree (the m_Polygons STL linked list) and compile a bounding box for each one.

The firsts section of the function (shown below) creates an STL list iterator (a PolygonList iterator) which it uses to step through the elements in the polygon list. For each polygon, it aliases its bounding box minimum and maximum extents vectors with local variables Min and Max (for ease of use) and then sets the bounding box of the polygon to initially bogus values.

```
void CBaseTree::CalculatePolyBounds( )
{
    ULONG                   i;
    CPolygon               *pCurrentPoly;
    PolygonList::iterator   Iterator;

    // Calculate polygon bounds
    for ( Iterator = m_Polygons.begin(); Iterator != m_Polygons.end(); ++Iterator )
    {
        // Get poly pointer and bounds references for easy access
        pCurrentPoly = *Iterator;
        if ( !pCurrentPoly ) continue;
        D3DXVECTOR3 & Min = pCurrentPoly->m_vecBoundsMin;
        D3DXVECTOR3 & Max = pCurrentPoly->m_vecBoundsMax;

        // Reset bounding box
        Min = D3DXVECTOR3( FLT_MAX, FLT_MAX, FLT_MAX );
        Max = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX );
```

Since FLT_MAX contains the maximum number that we can store in a float, and we set the maximum extents vector initially to the minimum possible value and the minimum extents vector to the maximum possible values, we create an initial huge “inside-out” box.

In the next section of the loop we test the position of each vertex in the polygon to see if it is contained within the box we have currently compiled. If not, the box will be adjusted to contain the vertex (this will always be the case for the first vertex due to the initial starting values of the box).

```
        // Build polygon bounds
        for ( i = 0; i < pCurrentPoly->m_nVertexCount; ++i )
        {
            CVertex & vtx = pCurrentPoly->m_pVertex[i];
            if ( vtx.x < Min.x ) Min.x = vtx.x;
            if ( vtx.y < Min.y ) Min.y = vtx.y;
            if ( vtx.z < Min.z ) Min.z = vtx.z;
            if ( vtx.x > Max.x ) Max.x = vtx.x;
            if ( vtx.y > Max.y ) Max.y = vtx.y;
            if ( vtx.z > Max.z ) Max.z = vtx.z;

        } // Next Vertex
```

Finally, before moving on to process the next polygon in the list, we will grow the box by 0.1 units in each direction to create a small buffer around the polygon. The reason we do this is to provide us a safety buffer when performing floating point tests. If two boxes (the polygon box and swept sphere box) were next to each other such that they were exactly touching, we risk the polygon being rejected for the narrow phase due to floating point accumulation/rounding errors. We certainly want to make sure that we do not reject a polygon from the narrow phase that we may be colliding with or we might be allowed to pass straight through it. By growing the box just slightly, we make this a fuzzy test where their boxes would overlap and the polygon would definitely be included in the case just mentioned.

```
        // Increase bounds slightly to relieve this operation during
        // intersection testing
        Min.x -= 0.1f; Min.y -= 0.1f; Min.z -= 0.1f;
        Max.x += 0.1f; Max.y += 0.1f; Max.z += 0.1f;

    } // Next Bounds
}
```

When this function, returns every polygon in the tree will contain a world space bounding box that the collision system (or any external component) can use to further refine the number of polygons in a leaf that get passed to the narrow phase.

14.11.6 Utility Methods - CBaseTree

These last two methods are not used by the application or by any other function in CBaseTree. They exist to make your life easier should you choose to implement the DebugDraw method in your derived classes. It is purely optional that you implement this method since ISpatialTree provides a default implementation that does nothing. Therefore, the application can always call this method even if you have not implemented it and no harm will be done.

The DebugDraw method will be implemented in all our derived classes in a very similar way. As discussed earlier, it will traverse the tree searching for visible leaf nodes. When a leaf node is found, we will draw a bounding box around the area represented by the leaf. We will also tint the screen red if the camera is currently contained in a leaf that contains polygon data. Since most of the code is geared towards generating and rendering the bounding box and handling the screen tinting, we decided to add the two functions that perform these tasks to CBaseTree (DrawBoundingBox and DrawScreenTint) and avoid duplicating code unnecessarily.

DrawBoundingBox – CBaseTree

The CBaseTree::DrawBoundingBox method will be called by the DebugDraw method in each of our derived classes (assuming the scene is calling ISpatialTree::DebugDraw). It will be passed the world space bounding box (the minimum and maximum extent vectors) of the leaf and (as its third parameter) the color we would like to render the wireframe box. The fourth parameter is a boolean that will indicate whether we would like the box to be rendered using the depth buffer.

When our DebugDraw routines call this method for a visible leaf which the camera is not currently in, the box will be rendered normally with depth testing enabled. For the current camera leaf we will pass false as this parameter and render it to the screen without depth testing. This ensures that it will always be rendered on top of anything already in the frame buffer, making it easier to see the box for the leaf we are currently standing in without it being obscured by nearby geometry.

The function has two static members. The first is an array of 24 vertices that will define the four vertices of each face of the cube (6 faces * 4 vertices = 24). We will also use a static boolean called BoxBuilt to indicate whether we have already built the box in the previous call. Since these are static methods they will retain their values each time the function is called. We do this so that the box mesh is only ever built the first time the method is ever called. If this method has never been called before, then the static BoxBuilt will be set to false and code will be executed to generate the box vertices and add them to the static vertex array. The BoxBuilt boolean will then be set to true so that the next time the method is called it will know that the box mesh has already been constructed and we will not have to do it again. Since the box we wish to draw will be a wireframe box, the vertices in this array will describe a list of line primitives which will be rendered using the D3DPT_LINELIST primitive type. The line list mesh will be stored in the vertices array as a 1x1x1 cube and it will be transformed and scaled to the size and position of the current leaf node. Let us look at this function in a few sections.

In the first section of the function we can see that a local structure is defined to describe what a box vertex looks like. Basically, it contains a position and a color. We can also see that if the tree does not currently store a pointer to a 3D device, then this is not a tree that is intended to be rendered and we return.

```
void CBaseTree::DrawBoundingBox( const D3DXVECTOR3 & Min,
                                 const D3DXVECTOR3 & Max,
                                 ULONG Color,
                                 bool bZEnable /* = false */ )
{
    struct BoxVert
    {
        D3DXVECTOR3 Pos;
        ULONG       Color;
    };

    ULONG          i;
    static BoxVert Vertices[24];
    static bool    BoxBuilt = false;
    D3DXMATRIX     mtxBounds, mtxIdentity;
    D3DXVECTOR3    BoundsCenter, BoundsExtents;
    ULONG          OldStates[6];

    // If there is no device, we can't draw
    if ( !m_pD3DDevice ) return;
```

Next we create an identity matrix (which will be used in a moment) and if the BoxBuilt boolean is false, we fill in the elements of the static vertex array. Notice that we position the 24 vertices such that they describe the vertices of each face in a unit sized cube; a cube of 1 unit in size in each direction which is centered at (0, 0, 0). This means the cube vertices are in the -0.5 to +0.5 range along each axis. We can think of this cube at this point as being in model space.

```
    // We need an identity matrix later
    D3DXMatrixIdentity( &mtxIdentity );

    // Build the box vertices if we have not done so already
    if ( !BoxBuilt )
    {
        // Bottom 4 edges
        Vertices[0].Pos  = D3DXVECTOR3( -0.5f, -0.5f, -0.5f );
        Vertices[1].Pos  = D3DXVECTOR3(  0.5f, -0.5f, -0.5f );

        Vertices[2].Pos  = D3DXVECTOR3(  0.5f, -0.5f, -0.5f );
        Vertices[3].Pos  = D3DXVECTOR3(  0.5f, -0.5f,  0.5f );

        Vertices[4].Pos  = D3DXVECTOR3(  0.5f, -0.5f,  0.5f );
        Vertices[5].Pos  = D3DXVECTOR3( -0.5f, -0.5f,  0.5f );

        Vertices[6].Pos  = D3DXVECTOR3( -0.5f, -0.5f,  0.5f );
        Vertices[7].Pos  = D3DXVECTOR3( -0.5f, -0.5f, -0.5f );

        // Top 4 edges
        Vertices[8].Pos  = D3DXVECTOR3( -0.5f,  0.5f, -0.5f );
        Vertices[9].Pos  = D3DXVECTOR3(  0.5f,  0.5f, -0.5f );

        Vertices[10].Pos = D3DXVECTOR3(  0.5f,  0.5f, -0.5f );
        Vertices[11].Pos = D3DXVECTOR3(  0.5f,  0.5f,  0.5f );

        Vertices[12].Pos = D3DXVECTOR3(  0.5f,  0.5f,  0.5f );
        Vertices[13].Pos = D3DXVECTOR3( -0.5f,  0.5f,  0.5f );

        Vertices[14].Pos = D3DXVECTOR3( -0.5f,  0.5f,  0.5f );
        Vertices[15].Pos = D3DXVECTOR3( -0.5f,  0.5f, -0.5f );

        // 4 Side 'Struts'
        Vertices[16].Pos = D3DXVECTOR3( -0.5f, -0.5f, -0.5f );
        Vertices[17].Pos = D3DXVECTOR3( -0.5f,  0.5f, -0.5f );

        Vertices[18].Pos = D3DXVECTOR3(  0.5f, -0.5f, -0.5f );
        Vertices[19].Pos = D3DXVECTOR3(  0.5f,  0.5f, -0.5f );

        Vertices[20].Pos = D3DXVECTOR3(  0.5f, -0.5f,  0.5f );
        Vertices[21].Pos = D3DXVECTOR3(  0.5f,  0.5f,  0.5f );

        Vertices[22].Pos = D3DXVECTOR3( -0.5f, -0.5f,  0.5f );
        Vertices[23].Pos = D3DXVECTOR3( -0.5f,  0.5f,  0.5f );

        // We're done
        BoxBuilt = true;

    } // End if box has not yet been built
```

At this point all 24 model space box vertices have been placed in the vertex array and the BoxBuilt boolean will be set to true. Thus, the above code will not be performed the next time this function is called to render another leaf box.

In the next section we loop through each of the 24 vertices and set their color value to the color parameter passed by the caller (the DebugDraw method of the derived class).

```
    // Set the color to that specified.
    for ( i = 0; i < 24; ++i ) Vertices[i].Color = Color;
```

In the next section we need to create a matrix that, when set on the device as a world matrix, will transform and scale the model space vertices such that the box is transformed into a box that is equal in size to the passed box vectors (the leaf's bounding box) and positioned in the world such that its center point is at the center point of the leaf.

First we need to find the world space center point of the leaf bounding box, which we can calculate by adding the leaf's world space minimum and maximum box vector and dividing the result by 2. We then calculate the size of the box (the diagonal size of the leaf's bounding box) by subtracting the minimum extent vector from the maximum vector.

```
    // Compute the bound's centre point and extents
    BoundsCenter  = (Min + Max) / 2.0f;
    BoundsExtents = Max - Min;
```

At this point BoundsExtents contains the diagonal length of the leaf bounding box. Since our box mesh is defined to be a 1x1x1 unit square (0.5 units in each dimension), if we store the BoundsExtents in the diagonal of a matrix, we will have a scaling matrix that will transform the vertices of the box mesh so that it becomes the same size as the leaf box in world space.

For example, let us imagine that the world space leaf bounding box extents are the two vectors (50, 50, 50) and (60, 60, 60). In this case, BoundsExtents will be:

$$(60, 60, 60) - (50, 50, 50) = (10, 10, 10)$$

If we store the x, y, and z components of the resulting vector in the diagonal of a matrix, we will get a matrix which will scale any x, y, and z vertex components by 10. Since the box is defined in the -0.5 to 0.5 range along each axis, we can see that when the box vertices are multiplied by this scaling matrix, it will result in vertices in the -0.5*10=-5 to 0.5*10=5 range (i.e., a box in the range of [-5, 5] along each axis). This is exactly as it should be as it matches the size of the leaf bounding box. All we have to do now is place the world space position vector of the center of the leaf (BoundsCenter) in the translation row of the matrix and we will have a world matrix that will properly scale and position the model space box mesh to match the leaf.

```
    // Build the scaling matrix
    D3DXMatrixScaling( &mtxBounds,
                       BoundsExtents.x,
                       BoundsExtents.y,
                       BoundsExtents.z );

    // Translate the bounding box matrix
    mtxBounds._41 = BoundsCenter.x;
    mtxBounds._42 = BoundsCenter.y;
    mtxBounds._43 = BoundsCenter.z;
```

```
    // Set the bounding box matrix
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxBounds );
```

At this point we have the world matrix set on the device and we are almost ready to render, but first we will need to set some render states. We will start by disabling Z buffer writing (we do not want our box lines to obscure real geometry) and enable or disable depth testing based on the boolean parameter passed in. We will also want to disable lighting (our lines are constructed from pre-colored vertices) and set the first texture/color stage so that only the diffuse color of the vertex is used. Of course, we had better retrieve and backup the current state settings because we would not want to change something that will cause the rest of the application to render incorrectly. So we will retrieve the current states that we intend to change and store them in a local states array (OldStates) as shown below.

```
    // Retrieve old states
    m_pD3DDevice->GetRenderState( D3DRS_ZWRITEENABLE, &OldStates[0] );
    m_pD3DDevice->GetRenderState( D3DRS_ZENABLE, &OldStates[1] );
    m_pD3DDevice->GetRenderState( D3DRS_LIGHTING, &OldStates[2] );
    m_pD3DDevice->GetRenderState( D3DRS_COLORVERTEX, &OldStates[3] );
    m_pD3DDevice->GetTextureStageState( 0, D3DTSS_COLORARG1, &OldStates[4] );
    m_pD3DDevice->GetTextureStageState( 0, D3DTSS_COLOROP, &OldStates[5] );
```

With the current states currently backed up we will setup the render states we wish to use to render our box edges.

```
    // Setup new states
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, bZEnable );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_COLORVERTEX, TRUE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_DIFFUSE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
```

With the pipeline now configured, we set the FVF of our box vertices and render. Since our box vertices contain positional information and a diffuse color we will inform the pipeline by using the D3DFVF_XYZ | D3DFVF_DIFFUSE flag combination. We will then render the vertices straight from the box vertex array using the DrawPrimitiveUP function. UP stands for ‘User Pointer’ and it allows us to render straight from system memory without having to store the vertices in vertex buffers. This is a highly inefficient way to render primitives, but it is acceptable in this case since it is only a debug routine and it makes the implementation a lot easier.

```
    // Draw
    m_pD3DDevice->SetFVF( D3DFVF_XYZ | D3DFVF_DIFFUSE );
    m_pD3DDevice->DrawPrimitiveUP( D3DPT_LINELIST, 12, &Vertices, sizeof(BoxVert));
```

As you can see in the above code, we ask DrawPrimitiveUP to draw 12 lines, where each line consists of two vertices (start and end points). Thus the vertex array we pass in as the third parameter must contain at least 12*2=24 vertices, which we know ours does. The fourth parameter is the stride of the vertex structure we are using so the pipeline knows how many bytes to advance when stepping from vertex to vertex during the transformation process.

With all lines rendered for the currently passed leaf node, we restore the render states we backed up earlier and reset the world matrix to identity.

```
    // Reset old states
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, OldStates[0] );
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, OldStates[1] );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, OldStates[2] );
    m_pD3DDevice->SetRenderState( D3DRS_COLORVERTEX, OldStates[3] );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, OldStates[4] );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, OldStates[5] );

    // Reset the matrix
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxIdentity );
}
```

DrawScreenTint – CBaseTree

This is utility function tints the screen a red color when the camera is positioned inside a populated leaf node. It accepts one parameter: a DWORD containing the color to tint the screen.

This technique should be familiar to you since we did the same thing in Chapter 7 of Module I when we implemented our underwater effect. We just create four transformed and lit vertices (screen space vertices with diffuse color components) and position them at the four corners of the viewport. We then render the quad to the screen with alpha blending enabled (and with the depth buffer disabled so that it is definitely rendered on top of everything else) to blend the color of the quad over the entire scene stored in the frame buffer..

The first part of the function is shown below. Note that we define a vertex structure (TintVert) which is laid out as a transformed and lit vertex. We know that when we pass a pre-transformed (viewport space) vertex to the pipeline, its positional vector is 4D, not 3D. The first two positional components of the vertex (x and y) describe the position on the viewport where the vertex is positioned. The third component (z) contains the depth buffer value in the range of 0.0 to 1.0. Since we are going to disable depth testing and writing when we render this quad, we will just set this to zero. The fourth positional component is called RHW by DirectX and should contain the reciprocal of homogenous W (i.e., 1 divided by the viewspace vertex Z component). This is all designed to work if you are performing your own transformation of geometry and merely wish DirectX to render the screen space polygons. 1/W gets closer to 1 the closer to the near plane the vertex is. It describes the depth of the vertex with respect to the camera. Recall that this value is used during rendering by the DirectX fog engine. However, we just wish to draw a quad on the screen without fog, so we will just set this to 1 which simulates a vertex very close to the near plane. However, since we are not using fog and the depth buffer is disabled, we could actually this value to anything without any ill effect.

```
void CBaseTree::DrawScreenTint( ULONG Color )
{
    struct TintVert
    {
        D3DXVECTOR4 Pos;
        ULONG       Color;
```

```
    };

    ULONG        i;
    TintVert     Vertices[4];
    ULONG        OldStates[10];
    D3DVIEWPORT9 Viewport;


    // If there is no device, we can't draw
    if ( !m_pD3DDevice ) return;

    // Retrieve the viewport dimensions
    m_pD3DDevice->GetViewport( &Viewport );


    // 4 Screen corners
    Vertices[0].Pos=D3DXVECTOR4((float)Viewport.X, (float)Viewport.Y, 0.0, 1.0f );

    Vertices[1].Pos=D3DXVECTOR4((float)Viewport.X + Viewport.Width,
                                (float)Viewport.Y, 0.0, 1.0f );

    Vertices[2].Pos  = D3DXVECTOR4( (float)Viewport.X + Viewport.Width,
                                    (float)Viewport.Y + Viewport.Height,
                                    0.0, 1.0f );

    Vertices[3].Pos  = D3DXVECTOR4( (float)Viewport.X,
                                    (float)Viewport.Y + Viewport.Height,
                                    0.0, 1.0f );

    // Set the color to that specified.
    for ( i = 0; i < 4; ++i ) Vertices[i].Color = Color;
```

The above code shows how we fetch the device viewport and retrieve the viewport rectangle. We will use this rectangle to position the vertices of our quad in the frame buffer. As you can see, the four vertices are positioned at the top left, top right, bottom right, and bottom left of the viewport. The z component of each vertex is set to 0.0 and the RHW component is set to 1.0. We then loop through each of the vertices and set its color properties to the color value passed into the function.

Next we will backup and set the render states and texture states we wish to use to render our screen effect. We will disable Z writing and testing, disable lighting, and enable alpha blending. We will set the source and destination color blending results to use the alpha component of the passed color to weight the blending between the color of the quad and the color already in the frame buffer. This means we need to set the source and destination blend render states to D3DBLEND_SRCALPHA and D3DBLEND_INVSRCALPHA and configure texture stage zero to take its alpha and color components from the diffuse vertex color. This will generate a final color for each pixel at the end of the texture stage cascade which has color and alpha components equal to the diffuse and alpha components of the passed color, respectively. We then pass this to the renderer where the alpha component will be used as the weighting factor with the blending modes we have configured.

Below we see the code that makes a backup of all the states we intend to change, sets all the new states, renders the quad, and then restores the original device states.

```
    // Retrieve old states
    m_pD3DDevice->GetRenderState( D3DRS_ZWRITEENABLE, &OldStates[0] );
    m_pD3DDevice->GetRenderState( D3DRS_ZENABLE, &OldStates[1] );
    m_pD3DDevice->GetRenderState( D3DRS_LIGHTING, &OldStates[2] );
    m_pD3DDevice->GetRenderState( D3DRS_COLORVERTEX, &OldStates[3] );
    m_pD3DDevice->GetRenderState( D3DRS_ALPHABLENDENABLE, &OldStates[4] );
    m_pD3DDevice->GetRenderState( D3DRS_ALPHATESTENABLE, &OldStates[5] );
    m_pD3DDevice->GetTextureStageState( 0, D3DTSS_COLORARG1, &OldStates[6] );
    m_pD3DDevice->GetTextureStageState( 0, D3DTSS_COLOROP, &OldStates[7] );
    m_pD3DDevice->GetTextureStageState( 0, D3DTSS_ALPHAARG1, &OldStates[8] );
    m_pD3DDevice->GetTextureStageState( 0, D3DTSS_ALPHAOP, &OldStates[9] );

    // Setup new states
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_COLORVERTEX, TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
    m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, FALSE );
    m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
    m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_DIFFUSE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1 );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_DIFFUSE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );

    // Draw
    m_pD3DDevice->SetFVF( D3DFVF_XYZRHW | D3DFVF_DIFFUSE );
    m_pD3DDevice->DrawPrimitiveUP(  D3DPT_TRIANGLEFAN,
                                    2,
                                    &Vertices,
                                    sizeof(TintVert) );

    // Reset old states
    m_pD3DDevice->SetRenderState( D3DRS_ZWRITEENABLE, OldStates[0] );
    m_pD3DDevice->SetRenderState( D3DRS_ZENABLE, OldStates[1] );
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, OldStates[2] );
    m_pD3DDevice->SetRenderState( D3DRS_COLORVERTEX, OldStates[3] );
    m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, OldStates[4] );
    m_pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, OldStates[5] );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, OldStates[6] );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_COLOROP, OldStates[7] );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, OldStates[8] );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, OldStates[9] );
}
```

We have now covered all the house keeping and utility building code from CBaseTree that needs to be covered in this lesson. The one exception is the Repair method that performs T-junction repair, but we will come back to that a little later in this lesson and devote an entire section to it. In the next chapter we will add a rendering system to this class, but for now we have everything we need in place to create our derived tree classes and use them to perform efficient collision queries.

Before we discuss T-junction repair and move on to look at the implementations of each of the derived classes, we will need to extend our CCollision library with a few more intersection routines that will be

used by the tree building and query functions in the derived classes. For example, we will need to be able to determine when a point is inside an AABB, when two AABBs are intersecting, and when a ray intersects an AABB. We will discuss the implementations of these functions next, and we will make them static members of the CCollision class so that they can be used even if the collision system is not being used.

14.12 Point/AABB Intersection

One of the simplest collision tests we can perform is determining whether a point is inside an AABB. Such a method is used to find out if a certain position is contained inside a leaf node's bounds, as one obvious example. Our DebugDraw routines will use this intersection test to determine whether the leaf currently being visited contains the camera position. If it does, then it renders the bounding box of the leaf a different color.

The test is easy and cheap to perform and involves nothing more than separately testing the x, y, and z components of the passed position vector to see if they fall between the minimum and maximum x, y and z extents of the AABB. If any component of the position vector is outside the range of its corresponding components in the AABB extent vectors, the point is not contained and we return false. That is, the test is essentially stating "If the x component of the point is between the minimum x extent and the maximum x extent of the box, we have passed the X axis test and the two other axes must be tested, etc."

The code to PointInAABB is just a few lines long. The function only returns true if none of the three axis tests fail.

```
bool CCollision::PointInAABB( const D3DXVECTOR3& Point,
                              const D3DXVECTOR3& Min,
                              const D3DXVECTOR3& Max,
                              bool bIgnoreX /* = false */,
                              bool bIgnoreY /* = false */,
                              bool bIgnoreZ /* = false */ )
{
    // Does the point fall outside any of the AABB planes?
    if ( !bIgnoreX && (Point.x < Min.x || Point.x > Max.x) ) return false;
    if ( !bIgnoreY && (Point.y < Min.y || Point.y > Max.y) ) return false;
    if ( !bIgnoreZ && (Point.z < Min.z || Point.z > Max.z) ) return false;

    // We are behind all planes
    return true;
}
```

Notice how this method not only accepts the point and the two AABB extents as parameters, it also accepts three booleans allowing us to instruct the function to ignore tests along any axes. For example, if we pass in true for the bIgnoreY parameter, the Y component of the point will not be tested against the Y components of the AABB extents (i.e., the function will behave as if the Y axis test has not failed and that the point always falls within the box with respect to the Y axis).

It is quite useful to have the ability to ignore certain components in the test. Indeed, you will see that this is a feature we provide in all of our AABB intersection tests. It is particularly handy when performing a collision query on a quad-tree. As discussed earlier, the vanilla quad-tree will always have leaf nodes of the same height (the max vertical range of the entire level). However, we also discussed (and will implement) the Y-variant quad-tree, where the vertical extents of each node fit the contained geometry. This can produce certain problems when querying the tree since there may be times when a given position is contained inside no leaf node. This is actually true of any tree type but is more easily demonstrated with the Y-variant tree.

For example, Figure 14.53 shows a terrain compiled into a Y-variant quad-tree. We also see a space ship hovering above the terrain.

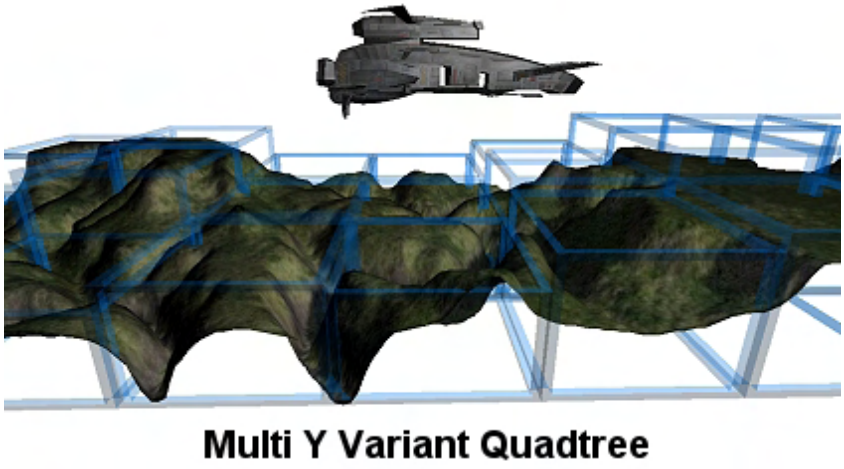


Figure 14.53

Imagine that our space ship has a spotlight mounted on the bottom of its hull, such that it illuminates the terrain immediately below it. In order to find the terrain polygons we need to illuminate, we would normally send the bounding box of the space ship down the tree and collect all the leaves it intersects. We can then fetch the polygons from those leaves and perform some custom lighting effect. However, in this case we can see that if we were to feed the bounding box of the ship into the tree, no leaves would be returned because it is not contained in any of the leaves. But, if we did the same test and ignored the Y component in each leaf (only testing the X and Z extents of the box), then the leaves beneath the ship would be returned even though the ship is not technically contained within their volume.

14.13 AABB/AABB Intersection

Another collision query that we will need to use quite a bit in our applications determines whether two AABBs intersect. This collision routine will be the heart of the CollectLeavesAABB method in our derived classes, which essentially traverses the tree with the passed query volumes and finds which leaves the passed bounding box intersects. At each leaf we will perform an AABB/AABB test and only add the leaf to the output list if there is an intersection and the collision function returns true.

Much like the PointInAABB test described above, the AABBIntersectAABB intersect function performs its tests on a per axis basis. If at any point we find an axis on which an overlap does not occur, the boxes do not overlap and the function can return false immediately. If there is overlap on each of the three axes, then the bounding boxes intersect one another and we return true. The test for an overlap on a given axis is very simple when dealing with AABBs as Figure 14.54 shows.

Figure 14.54

Figure 14.54 shows the X axis overlap test for two boxes that do indeed overlap. Since we are dealing with one axis at a time, the two boxes can be treated component-wise as lines on those axes. If Min1 and Max1 are the extent vectors of the first box and Min2 and Max2 are the extent vectors of the second box, we can see that if Min1.x is smaller than Max2.X and Max1.x is larger than Min2.x, then the boxes overlap on that axis. You should be able to imagine that if we were to move the blue line to the left so that they no longer overlapped, min2.x would no longer be smaller than Max1.x and the test would fail. That is, we would not have an overlap on the X axis which means the boxes cannot possibly be intersecting one another. When this is the case we can return false immediately without performing any other axis overlap tests.

The code is shown below. Note that it also supports the axis ignore functionality discussed in the previous function. If we choose to ignore any axis, then those axes are assumed to be overlapping.

```
bool CCollision::AABBIntersectAABB( const D3DXVECTOR3& Min1,
                                    const D3DXVECTOR3& Max1,
                                    const D3DXVECTOR3& Min2,
                                    const D3DXVECTOR3& Max2,
                                    bool bIgnoreX /* = false */,
                                    bool bIgnoreY /* = false */,
                                    bool bIgnoreZ /* = false */ )
{
    return (bIgnoreX || Min1.x <= Max2.x) && (bIgnoreY || Min1.y <= Max2.y) &&
           (bIgnoreZ || Min1.z <= Max2.z) && (bIgnoreX || Max1.x >= Min2.x) &&
           (bIgnoreY || Max1.y >= Min2.y) && (bIgnoreZ || Max1.z >= Min2.z);
}
```

The parameter list to the function includes the minimum and maximum extent vectors of the first AABB to test followed by the minimum and maximum extent vectors of the second AABB to test. Following this are three optional axis ignore booleans allowing us to ignore any of the axes during the test.

We have decided to also provide an overloaded version of this function that accepts a boolean reference as its first parameter that can be used to communicate back to the caller whether or not box 2 was completely contained inside box 1. This makes the test slower since it has to perform the containment test first, followed by the intersection test described above. However, there are times when dealing with

hierarchy traversal that this can prevent many future AABB/AABB intersection tests from needing to be performed.

For example, imagine that our application called the CollectLeavesAABB function for one of our derived tree classes. We know this function has the task of stepping through the tree and at each node testing for an intersection between the passed AABB and the AABB of the node/leaf. If the query AABB does not intersect a node's AABB then we do not have to bother traversing into any of its children, allowing us to reject portions of the tree from having to be testing and collected. However, if the query volume does intersect the node volume, we will want to step into its children. Whenever a leaf node is reached, we add its leaf structure to a list that can be passed back to the caller. However, with this new overloaded version of the function we can perform a optimization to our CollectLeavesAABB routine. If we know that the bounding box of a node is completely contained inside the query volume, then we already know that all of its child nodes (including the leaf nodes) will be contained in it also. Thus, we do not need to perform any further AABB intersection tests on any of its child nodes. Instead we can just traverse into its children searching for leaf nodes which, once found, are immediately added to the leaf list that will be returned.

The overloaded function is shown below with the containment tests performed at the beginning of the function. The passed bContained boolean is initially set to true, meaning that box 2 is fully contained in box 1 and each of the containment tests tries to find proof that box 2 pierces the bounds of box 1 and is therefore not fully contained. This is once again done of a per axis basis, so for example, when testing the X axis for containment, but we are essentially just trying to prove that either the minimum extents of box 2 are smaller then the minimum extents of box 1 or that the maximum extents of box 2 are larger than the maximum extents of box 1. If this is true for any axis, then box 2 is not contained in box1 and the containment boolean is set to false. When this is the case, we must perform the intersection test described above to test if the boxes even intersect. The intersection test at the end of the function only has to be performed if the containment test failed. As soon as we have proof that box 2 is contained in box 1 we can return true for intersection with the bContainment boolean set to true.

```
bool CCollision::AABBIntersectAABB( bool & bContained,
                                    const D3DXVECTOR3& Min1,
                                    const D3DXVECTOR3& Max1,
                                    const D3DXVECTOR3& Min2,
                                    const D3DXVECTOR3& Max2,
                                    bool bIgnoreX /* = false */,
                                    bool bIgnoreY /* = false */,
                                    bool bIgnoreZ /* = false */ )
{
    // Set to true by default
    bContained = true;

    // Is box contained totally inside
    if ( !bIgnoreX && (Min2.x < Min1.x || Min2.x > Max1.x) ) bContained = false;
    else
    if ( !bIgnoreY && (Min2.y < Min1.y || Min2.y > Max1.y) ) bContained = false;
    else
    if ( !bIgnoreZ && (Min2.z < Min1.z || Min2.z > Max1.z) ) bContained = false;
    else
    if ( !bIgnoreX && (Max2.x < Min1.x || Max2.x > Max1.x) ) bContained = false;
    else
    if ( !bIgnoreY && (Max2.y < Min1.y || Max2.y > Max1.y) ) bContained = false;
```

```
    else
    if ( !bIgnoreZ && (Max2.z < Min1.z || Max2.z > Max1.z) ) bContained = false;

    // Return immediately if it's fully contained
    if ( bContained == true ) return true;

    // Perform full intersection test
    return (bIgnoreX || Min1.x <= Max2.x) && (bIgnoreY || Min1.y <= Max2.y) &&
           (bIgnoreZ || Min1.z <= Max2.z) && (bIgnoreX || Max1.x >= Min2.x) &&
           (bIgnoreY || Max1.y >= Min2.y) && (bIgnoreZ || Max1.z >= Min2.z);
}
```

14.14 Ray/AABB Intersection

We have talked a lot about the usefulness of being able to send a ray through the tree and get back a list of intersecting leaves. One example of where such a technique will be used is in Module III when we will write a lightmap compiler. Beams of light emanating from a light source will be modelled as rays and we will need to determine as quickly as possible whether any polygons in the scene block that ray. As our scenes will likely involve of tens of thousands of polygons or more, we will be performing literally millions of these tests. To make sure that our lightmap compiler will compile its texture data as quickly as possible, the scene will first be compiled into a spatial tree so that we can query the scene efficiently by performing the ray query on the tree. There are plenty of other examples, but this should give you further indication that spatial partitioning will be an important tool in many areas.

When examining the ISpatialTree interface we saw that our derived tree classes will be expected to implement a method called CollectLeavesRay. This method will be passed a ray and will traverse the tree testing which leaves the ray intersects. These leaves will then be returned so that the calling application can access the polygon data. In the lightmap compiler discussed above, the leaves returned will contain the polygons that have the potential to block the ray. This handful of polygons can then be tested more closely using ray/polygon intersection tests.

The heart of the CollectLeavesRay function (at least for the derived classes we implement) will be the testing of a ray to see if it intersects the bounding box of a given node. If the ray does intersect a node then we have to traverse into its children and continue testing. If it does not, then we can abandon that branch of the tree and all the leaf data it contains. Whenever we enter a leaf node, the leaf object will be added to a list that will be returned to the caller. Thus, the function will return to the application a list of the leaves the ray intersected. It is clear then that we must learn how to test for an intersection between a ray and an axis aligned bounding box.

14.14.1 The Slabs Method

We will be using the slabs method for intersection testing since it will works for both axis aligned bounding boxes and oriented bounding boxes. Our implementation of this method however will be optimized to take advantage of the fact that we will only be using it for AABB testing. As such, our

function is to be used only for the testing of AABBs. Although we will be implementing a faster AABB only method, the intersection theory we discuss here should make it a fairly simple matter for you to implement a version of the function that works with OBBs as well.

The slabs method works by calculating the intersection between the ray and each set of parallel planes described by the cube faces. A slab is therefore a pair of faces that are parallel to one another. If we think of an axis aligned bounding box, it would be comprised of three slabs. The first slab might be the pair of planes described by the front and back face of the cube, whose normals are aligned to the coordinate system Z axis. The second slab might be the planes of the right and left faces whose normals are aligned with the coordinate system X axis. The third slab would be comprised of the planes of the top and bottom faces whose normals are aligned with the Y axis of the coordinate system.

The test can easily be converted to two dimensions just by dropping tests against the slab whose planes are aligned with the coordinate system Z axis. To make visualizing the slabs method a little easier, we will discuss the two dimensional case in our images. The only difference is that in the three dimensional case, we are testing three slabs instead of two.

Figure 14.55 shows the minimum and maximum extent vectors of an AABB which describe the bottom left and top right corners of a 2D axis aligned bounding box. It also shows the ray that we would like to test for intersection against the bounding box. As can be clearly seen, the ray does intersect the box; we just need some way of determining this.

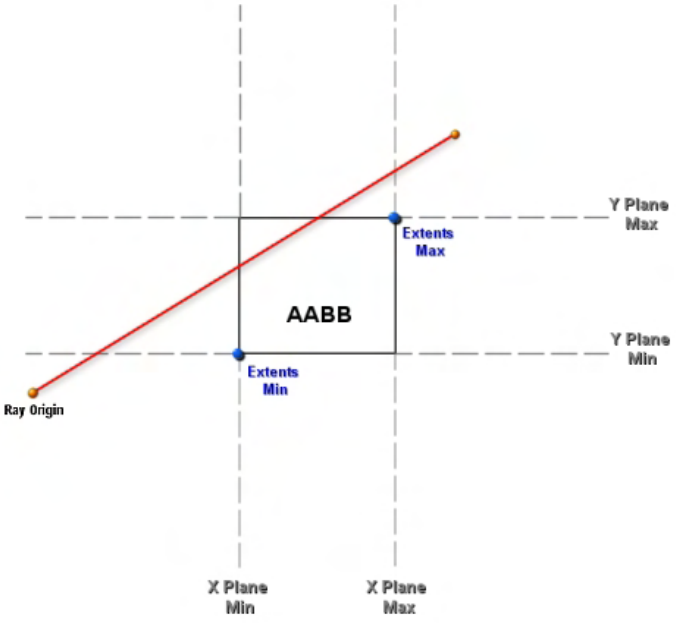


Figure 14.55

We can see that the point at the bottom left corner of the AABB (Extents Min) describes a point that is on the plane of the left face of the box. This is a plane whose normal is aligned with the coordinate system X axis. The top right corner of the box (Extents Max) describes a point that is on the same plane as the right face of the box and whose normal is also aligned to the coordinate system X axis. These two planes are parallel to one another and share the same normal, and as such, comprise a slab. This is the slab for the X axis and it contains the planes labelled ‘X Plane Min’ and ‘X Plane Max’. They are assigned the min and max names based on their distance to the ray origin; that is, based on the time of intersection between the ray and that plane. X Plane Min for example will intersect the ray closer to the ray origin, so it is called the minimum plane of the slab. We can see that a slab bounds the AABB along one axis. In this case we are describing the X slab which bounds the AABB on its left and right sides. The reason why this is important will become clear in a moment.

If we look at the same diagram (Figure 14.55) we can see that the same two extent vectors of the AABB also describe points that are on two planes aligned with the coordinate system Y axis, and thus describe

the Y slab that bounds the box on its top and bottom sides. Thus, Extents Min is also on the Y Plane Min plane and Extents Max is also on the Y Plane Max plane. Therefore, these two extents vectors actually describe four planes and two slabs. The planes of the Y slabs would each have normals aligned with the coordinate system Y axis and are once again labelled using min and max based on their distance from ray origin (the *t* value at which the ray would intersect them).

This means that we have all the information we need to define each plane in each of the slabs. We know the plane normals used by each slab since they will be aligned with the coordinate system axis. That is, one slab will have normals <1,0,0> and the other will have planes with normals <0,1,0>. In the 3D case there will be a third slab consisting of two planes with the normal <0,0,1>. Of course, we know that a normal is not enough to describe a plane since there are an infinite number of planes that share the same normal but are positioned at different distances from the origin. However, we have the points that are on each of the four planes (the two extent vectors) so we have everything we need to test the ray for intersection against each of these four planes.

The test is very simple. For each slab, we perform an intersection test between the ray and each of the planes comprising that slab. This will return us two *t* values of intersection. For example, we might process the X slab first, which would mean performing two ray/plane intersection tests between the ray and each plane in the slab (X Plane Min and X Plane Max). Once we get back these *t* values, we store them in two variables called Min and Max, sorted by value. As we test each slab and retrieve the minimum and maximum *t* values (for each plane) we compare them against two values that are keeping a record of the highest minimum *t* value and the lowest maximum *t* values found so far. If the minimum *t* value we have just calculated for the current slab is higher than the highest minimum *t* value we have found for a previous slab, we overwrite it with the new *t* value. Likewise, if the maximum *t* value for the current slab is lower than the lowest maximum *t* value we have recorded from all previously tested slabs, we overwrite the value with this new lowest maximum *t* value.

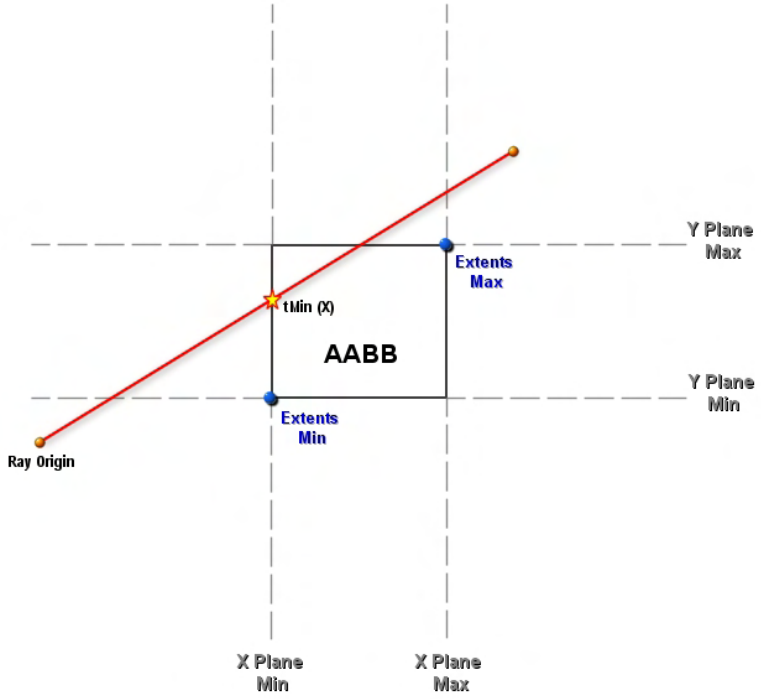


Figure 14.56

After all slabs have been evaluated, we will have two variables that contain the lowest maximum *t* value and the highest minimum *t* value we have found. If the highest minimum *t* value is larger than the lowest maximum *t* value then the ray does not intersect the box.

That might sound a little complicated, so let us have a look at an example that shows us computing the *t* values for each slab, one step at a time. In this first example, the ray does intersect the box, so the highest minimum *t* value we find for all the slabs must be lower than the lowest maximum *t* value we found from any slab according to the statement we just made a moment ago.

Let us step through the process of calculating the *t* values for the X slab first.

In Figure 14.56 we start with the X slab and calculate the *t* value of intersection between the ray and the plane that contains the minimum extents vector with a normal aligned to the coordinate system X axis (X Plane Min). The *t* value returned is shown on the diagram as tMin(x). This is the point at which the ray intersects the first plane of the X slab. Although we have already called this the minimum plane, we do not always know that this is the case. If the ray was intersecting the box from the right hand side instead, the second plane in the slab (X Plane Max) would be the first (minimum) plane of intersection.

Next we calculate the *t* value for the other plane of the slab (see Figure 14.57). This is the plane that contains the point ExtentsMax and shares the same normal (1,0,0).

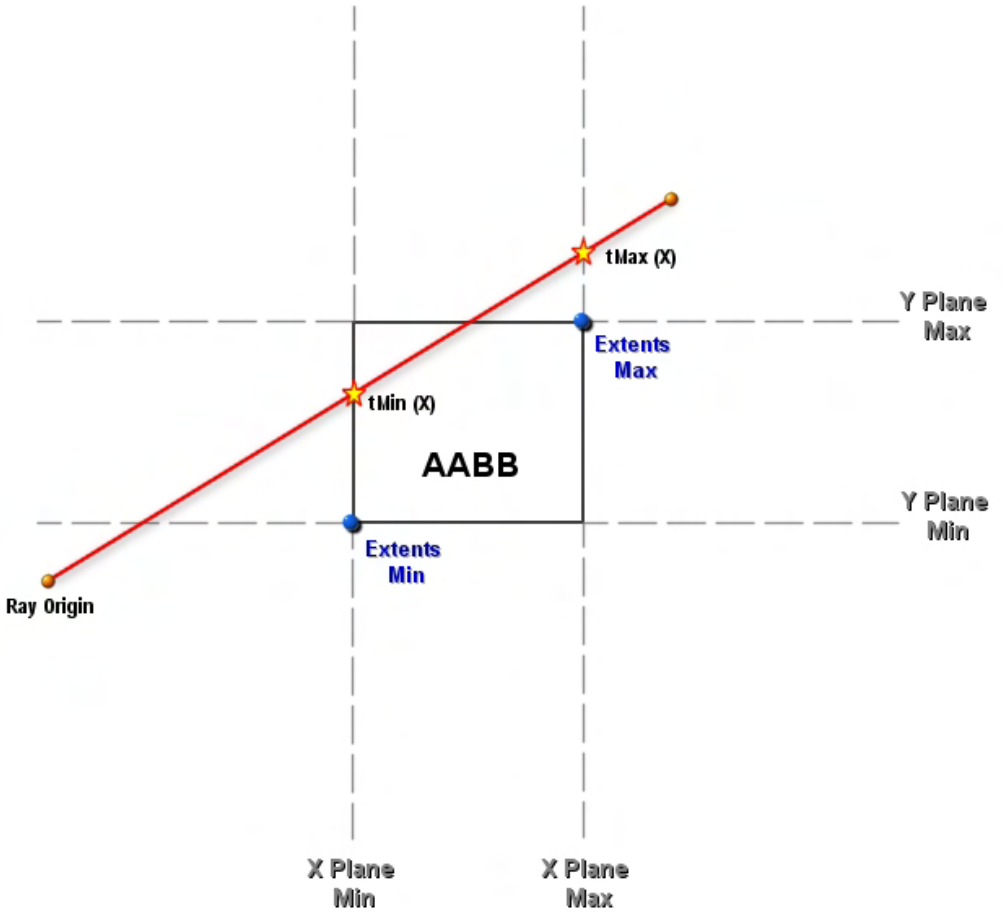


Figure 14.57

At this point we have the two *t* values for this slab, so we test which is greater and store them in the temporary variables tMin(x) and tMax(x). That is, we store in tMin(x) the plane in the slab that the ray intersects first (the smallest *t* value). In this example, we actually calculated the *t* values in this order but if the ray was intersecting from the right side of the cube, tMin(x) would contain the intersection with the X Max plane instead, and vice versa since the intersection order would be reversed.

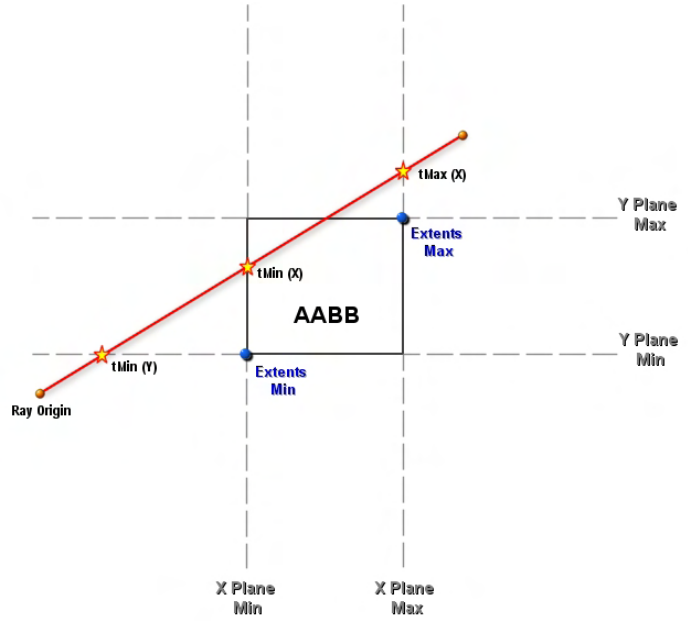


Figure 14.58

Now it is time to process the next slab (the Y slab). First we calculate the point of intersection between the ray and the first plane in the Y slab (Y Plane Min). This is shown in Figure 14.58 and is labelled tMin(y).

Our next task is to calculate another ray plane intersection test between the ray and the second plane in the Y slab (Y Plane Max). The point of intersection between the ray and second plane in the Y slab is shown in Figure 14.59 and is labelled tMax(y).

Once we have both *t* values for the Y slab, we sort them into two temporary variables as before so that tMin stores the lowest *t* value in the slab and tMax holds the highest *t* value in the slab.

The next bit of logic makes it all work. Once we have calculated the two *t* values for a slab, we see if the minimum *t* value for that slab is greater than the largest minimum *t* value we have found so far. If so, we record it. We can see in Figure 14.59 that the highest minimum *t* value from both slabs is tMin(x), which was the minimum *t* value we recorded for the first slab. In other words, this is the minimum *t* value that intersects the ray furthest from the origin compared to any other minimum *t* value calculated for other slabs. Also, when we get the maximum *t* value for a slab, we test to see if it is lower in value than any *t* value we have found so far. If it is, then we record it. At the end of testing all slabs we will have the lowest maximum *t* value found and the lowest minimum *t* value found.

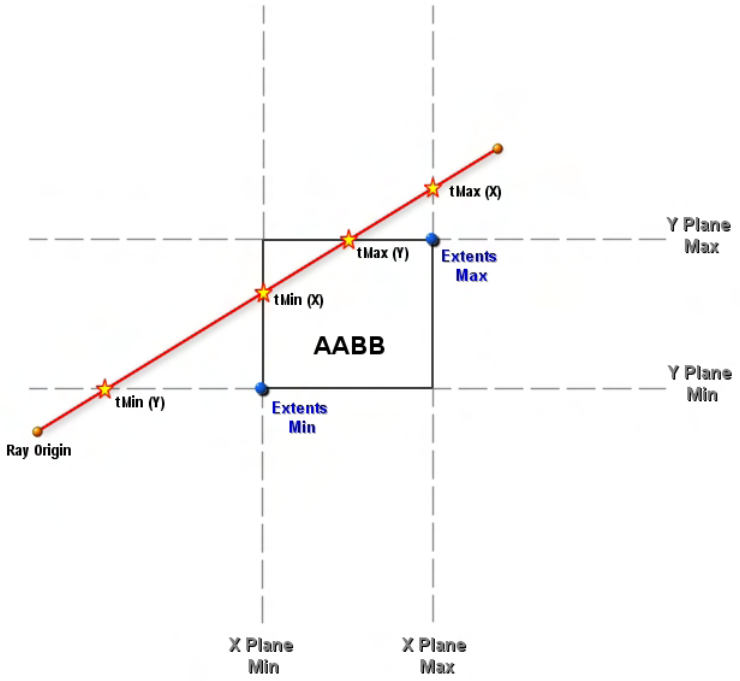


Figure 14.59

In Figure 14.59 we can see that when we calculate the maximum *t* value for the Y slab (tMax(y)), this intersection happens closer to the ray origin than the maximum *t* value found for the previous slab (yMax(x)) so we record that this is the lowest maximum *t* value we have found so far.

At the end of testing each slab we will have recorded the highest minimum value and the lowest maximum value as shown below.

tMin = tMin(X)
tMax = tMax(Y)

tMin and tMax were the temporary variables used to record the highest minimum *t* value and the lowest maximum *t* value during slab testing. We have highlighted the tMin and tMax values in Figure 14.60 below. As tMin is smaller than tMax, this means the ray intersects the box and we can return true for intersection.

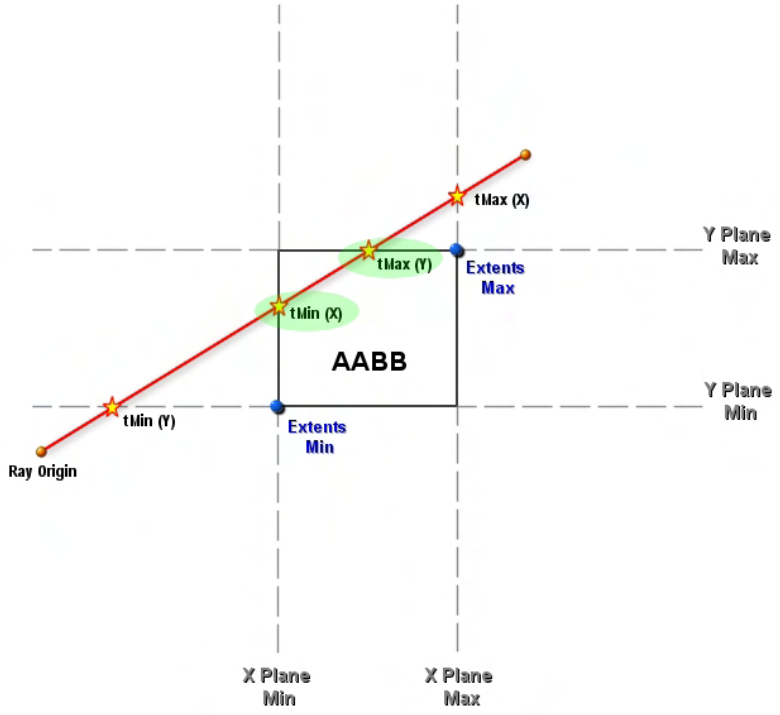


Figure 14.60

To understand why this means an intersection took place, imagine taking the red ray in the above diagram and moving it diagonally up and to the left.

As you imagine slowly moving it, you should be able to see that tMin(X) would move up its plane and tMax(Y) would move left along its plane and they would be identical values at the point where the ray just touches the top left corner. If we continue to move the ray up and to the left we can see that tMin(X) and tMax(Y) would swap around and the intersection with tMax(Y) would happen before the intersection with tMin(X). Thus we have a situation where the highest minimum *t* value is greater than the lowest maximum *t* value and thus, we have no intersection.

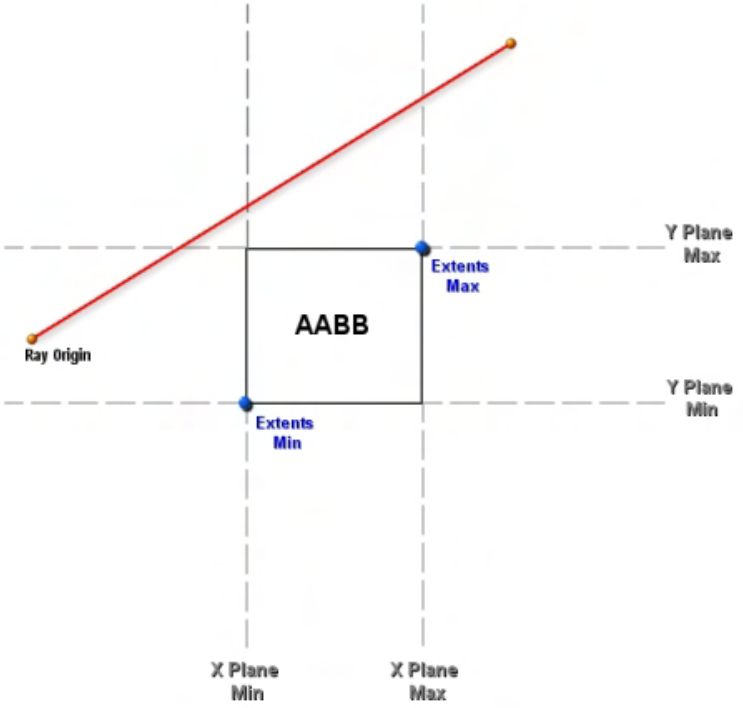


Figure 14.61

Let us try this out to make sure we are correct. Figure 14.61 shows a new example where the ray does not intersect the box. We will now quickly step through the same process and we should find that tMin is larger than tMax at the end of testing all the slabs.

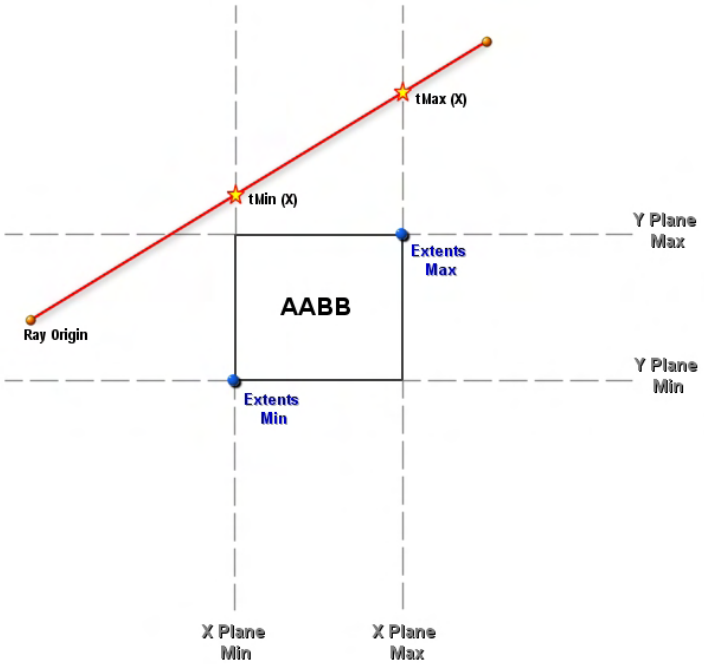


Figure 14.62

As before, we process each slab one at a time. Figure 14.62 shows the results of intersection testing the ray with the two planes comprising the X slab. Again, we test the two returned *t* values and sort them so that tMin(X) holds the first point of intersection and tMax(X) stores the second point of intersection.

Obviously, because we have only tested one slab at this point, the highest minimum *t* value we have found is tMin(X) and the lowest maximum *t* value we have found is tMax(X).

Next we move on to the Y slab and calculate the minimum and maximum intersection *t* values for both planes of that slab. These are shown in Figure 14.63 as tMin(y) and tMax(y). Notice this time however that the lowest maximum value we have found is tMax(y) which intersects the ray closer to the origin than the highest minimum *t* value (tMin(x)). As such, tMin > tMax and we know that the ray does not intersect the box. The *t* values that would end up in tMin and tMax at the end of testing all slabs are highlighted green in Figure 14.63.

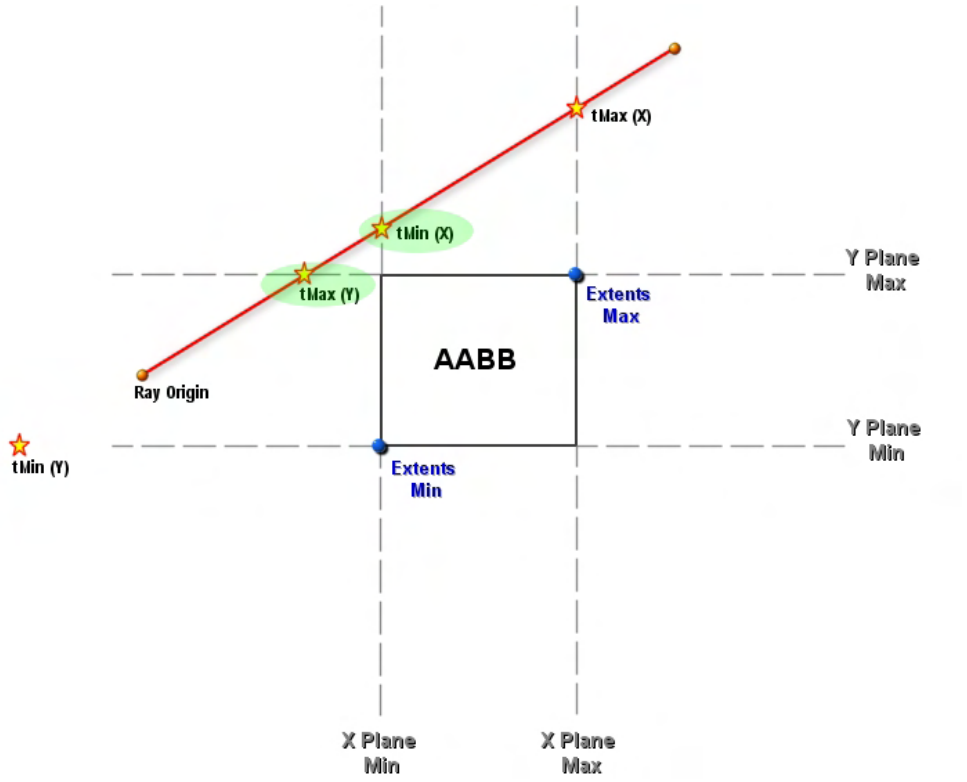


Figure 14.63

Notice that in this second example (Figure 14.63) the minimum *t* value for the Y slab (tMin(y)) is actually behind the ray origin and will thus be a negative value. This works fine since we are only interested in recording the highest minimum value found so far (so this *t* value would therefore be ignored).

Now it might not be clear why, but the process we have just described work for both OBBs and AABBs. Regardless of the orientation of the planes, as long as we have the two corner points of the box, we have points that lay on all slab planes and calculating the *t* values for each slab can be done using the CCollision::RayIntersectPlane method discussed in the previous chapter.

Next we will discuss some optimizations that can be performed that will allow our code to perform the plane intersections much quicker with axis aligned bounding boxes. Unfortunately, this will come at the cost of having a function that will not work with OBBs. However, you should be able to write your own Ray/OBB code should you need it based on the above theory.

To understand the optimizations that can be made in the case of an AABB, we will concentrate on calculating the *t* values for just one of the slabs. We will focus on the X slab for now and initially concentrate on calculating the *t* value for just one of its planes. Figure 14.64 shows the earlier example of a ray intersecting an AABB and the first plane in the X slab we need to calculate the *t* value for. As we can see, ExtentsMin describes a point on this plane and the plane normal is assumed to be aligned with the coordinate system X axis <1,0,0>.

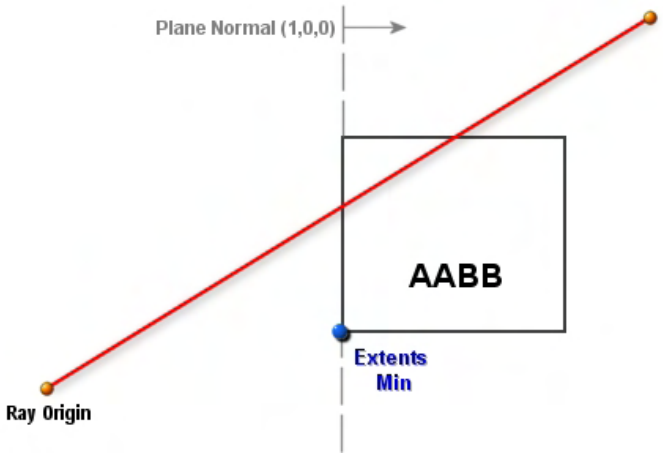


Figure 14.64

In order to find the point of intersection along the ray we first need to find the distance to the plane from the ray origin. We would normally do this by subtracting from the point known to be on the plane (ExtentsMin) from the ray origin, giving us the vector EV1 shown in Figure 14.65.

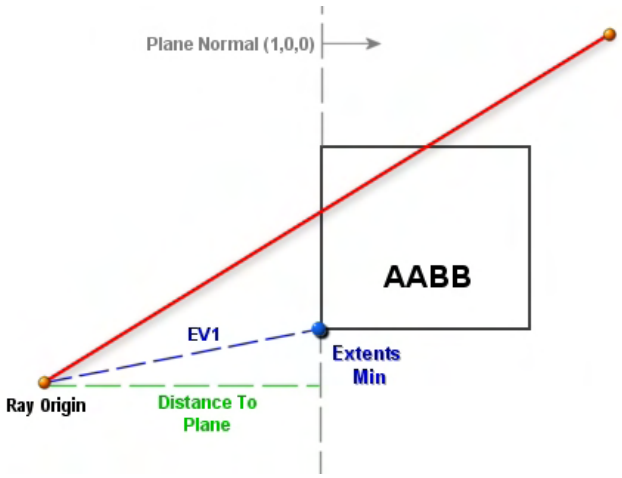


Figure 14.65

Once we have the vector EV1 we would dot it with the plane normal. Since the plane normal is unit length and the two vector origins are brought together for the dot product operation, this will scale the length of vector EV1 by the cosine of the angle between the two vectors, returning a single scalar value that describes the length of vector EV1 projected along the direction of the plane normal (the length of the green dashed line shown in Figure 14.65). This is the distance to the plane. However, why bother doing a dot product between the normal and the vector EV1 when we know that two of the components of an axis aligned plane normal will always be zero and the other will always be one? In fact, we do not have to perform a dot product between EV1 and the normal to find the distance to the plane; it is simply the value stored in EV1's x component, as shown below.

Distance To Plane **= EV1 ● Normal**
= EV1.x*Normal.x + EV1.y*Normal.y + EV1.z*Normal.z
= EV1.x*1 + EV1.y*0 + EV1.x*0
= EV1.x*1
= EV1.x

Once we have calculated vector EV1, its x component will tell us the distance to the first plane in the slab. We have just saved ourselves three multiplies and two additions per plane test.

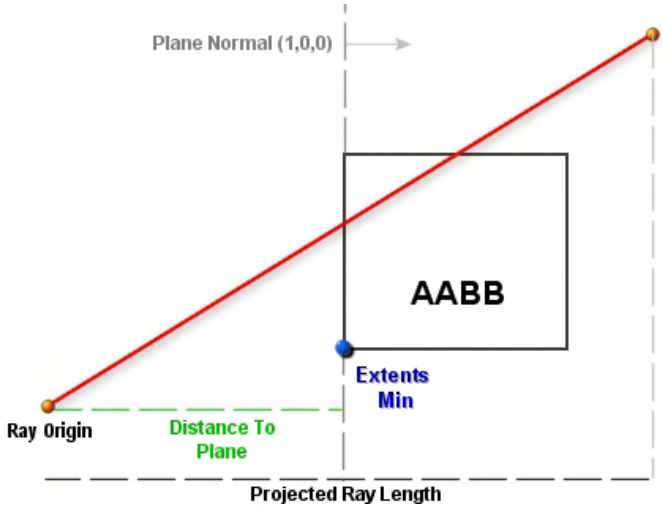


Figure 14.66

After we have found the distance from the ray origin to the plane, the next step is to project the ray length along the plane normal as shown in Figure 14.66. The ray delta vector is a vector describing the ray's direction and length from the ray origin to its end point. If we dot this with the plane normal, we will project it onto the line of the plane normal which will tell us the projected ray length. This is the length of the black dashed line at the bottom of Figure 14.66. Note that it tells us the length of the adjacent side of a triangle with the ray delta vector as the hypotenuse and the plane normal describing the direction of the adjacent side. Because we have projected the ray length along the plane normal, we can just divide the distance to the plane from the ray origin by the projected ray length to get the *t* value of intersection. In the above diagram it would be somewhere near to 0.5 as the plane intersects the ray roughly halfway along. In the last chapter we learned that we can calculate the *t* value of intersection between a ray and a plane as follows:

$$t = \frac{\text{Distance To Plane}}{V \bullet N}$$

$$t = \frac{EV1.x}{V \bullet N}$$

Where V is the ray delta vector and N is the plane normal. Once again, because we know that the plane normal of axis aligned plane will be a unit length vector with two of its components set to zero, the full dot product is not needed. This simplifies the denominator in the above equation to the following (for the X axis):

V ● N **= V.x*Normal.x + V.y*Normal.y + V.z*Normal.z**
= V.x*1 + V.y*0 + V.x*0
= V.x*1
= V.x

The final equation for calculating the *t* value for the ray and the plane shown in the above diagrams (the first plane of the X slab) has been simplified to the following:

$$t = \frac{EV1.x}{V.x}$$

We can do exactly the same for the second slab in the X plane, only this time we calculate EV1 by subtracting the ray origin from Extents Max instead of Extents Min.

This same logic works for all slabs that are axis aligned. Thus, to calculate the *t* values for the Y slab:

$$t = \frac{EV1.y}{V.y}$$

When we introduce the third dimension and a new Z aligned slab:

$$t = \frac{EV1.z}{V.z}$$

Remember that for each plane, we must calculate EV1 by subtracting the ray origin from either ExtentsMin or ExtentsMax depending on whether we are intersection testing the minimum or maximum plane of the slab.

Let us now look at the code to our new RayIntersectAABB function. The function is passed the ray origin and delta vector (Velocity) and is also passed the minimum and maximum extents of the AABB. As the fourth parameter we pass a variable by reference that will contain the *t* value of intersection between the ray and the box on function return. As with all of our AABB routines, we also offer the option to ignore any axis (slab) test. This allows us to (for example) consider a ray to intersect a box if it passes through the box in the X and Z dimensions but is above or below the box (useful when running queries against a Y-variant quad-tree).

```
bool CCollision::RayIntersectAABB(  const D3DXVECTOR3& Origin,
                                    const D3DXVECTOR3& Velocity,
                                    const D3DXVECTOR3& Min,
                                    const D3DXVECTOR3& Max,
                                    float& t,
                                    bool bIgnoreX /* = false */,
                                    bool bIgnoreY /* = false */,
                                    bool bIgnoreZ /* = false */ )
{
    float tMin  = -FLT_MAX, tMax =  FLT_MAX, t1, t2, fTemp;
    D3DXVECTOR3 ExtentsMin, ExtentsMax, P, RecipDir;
    ULONG       i;

    // Calculate required values
```

```
    ExtentsMin  = Min - Origin;
    ExtentsMax  = Max - Origin;
    RecipDir = D3DXVECTOR3( 1.0f/Velocity.x, 1.0f/Velocity.y, 1.0f/Velocity.z);
```

For efficiency we will be testing one slab at a time (two planes simultaneously), so we subtract the ray origin from both the minimum extents and maximum extents of the box. In this code, we can think of the variable ExtentsMin as describing the vector EV1 in our diagrams (the vector from the ray to the minimum plane in the slab) and we can think of ExtentsMax as describing a vector from the ray origin to the point on the second plane in the slab. At this point, the ExtentsMin <x, y, z> components contain the distances from the ray origin to the three minimum planes of each slab and ExtentsMax contains the three distances from the ray origin to the second plane in each slab. As we discussed earlier, we will calculate the *t* value by dividing the component in the ExtentsMin and ExtentsMax vectors that matches the plane we are currently testing by the matching component in the ray delta vector (Velocity). As we may have to test six planes in all, this means six divisions. Since divisions are slower than multiplications, we perform three divisions to create a reciprocal delta vector which we can then multiply with this vector. This will have the same effect as dividing by the ray's delta vector.

Now it is time to test each slab by setting up a loop to count three times. Each iteration of this loop will calculate the two t values from the planes forming that slab. As you can see below, if the slab we are currently testing is one we have chosen to ignore, we just skip any processing and advance to the next iteration.

```
    // Test box 'slabs'
    for ( i = 0; i < 3; ++i )
    {
        // Ignore this component?
        if ( i == 0 && bIgnoreX ) continue;
        else if ( i == 1 && bIgnoreY ) continue;
        else if ( i == 2 && bIgnoreZ ) continue;
```

If we get this far then this is a slab we wish to process. If we are on iteration zero then we are processing the X slab. If the matching component in the ray delta vector is zero (with tolerance) then the ray is running parallel to the current slab and could not intersect any of its planes, so we skip this slab. Therefore, the *t* value calculations for each plane in the current slab are only executed when the ray's delta vector does not have a zero in the component that matches the slab currently being tested. So if Velocity.x=0 then there is no point trying to calculate a point of intersection with the plane's comprising the X slab as the ray does not get any closer or further from the slab with distance.

```
        // Is it pointing toward?
        if ( fabsf(Velocity[i]) > 1e-3f )
        {
```

If we get inside this code block then we need to calculate the two *t* values for the planes of this slab. As discussed, this is a simple case of dividing the matching components in each of the extents vectors by the matching component in the ray delta vector.

```
            t1 = ExtentsMax[i] * RecipDir[i];
            t2 = ExtentsMin[i] * RecipDir[i];
```

At this point we have the *t* values between the ray and the planes of this slab stored in t1 and t2. We want to make sure that t1 holds the smallest *t* value, so we swap their values if this is not the case.

```
            // Reorder if necessary
            if ( t1 > t2 ) { fTemp = t1; t1 = t2; t2 = fTemp; }
```

Next we test to see if t1 (the minimum *t* value) is greater than any other minimum *t* value we have stored so far (tMin). If so, we overwrite its value with the new *t* value. Also, if the maximum *t* value (t2) is less than any maximum *t* value we have found so far (tMax), we also record its value.

```
            // Compare and validate
            if ( t1 > tMin ) tMin  = t1;
            if ( t2 < tMax ) tMax = t2;
```

At this point tMin will contain the highest minimum *t* value found so far for all previous slabs, and tMax will contain the lowest maximum *t* value. If we find (even if we are testing only the first slab) that tMin is ever greater than tMax, it means the ray cannot possibly intersect the box and we can return false immediately without testing any other slabs. Also, if we ever find that tMax is smaller that zero, then it means the box is behind the ray and we can also return false immediately. Here is the rest of the slab testing loop.

```
            if ( tMin  > tMax ) return false;
            if ( tMax < 0 ) return false;

        } // End if toward

    } // Next 'Slab'
```

If we reach this point in the function without returning then we know we have an intersection between the ray and box. All we wish to do now is return the first point of intersection. This should always be the value contained in tMin, unless it is a negative value, in which case the plane is behind the ray origin and tMax should be returned.

```
    // Pick the correct t value
    if ( tMin > 0 ) t = tMin; else t = tMax;

    // We intersected!
    return true;
}
```

That was quite a lot of explanation for what turned out to be a very small function. However, this is exactly the sort query you may have to explain at a job interview so it is good that we explored it fully.

14.15 AABB/Plane Intersection

The algorithm for classifying an AABB against a plane is not really new to us since it is something we have been doing since we first introduced frustum culling in Module I. You will recall that we introduced code to detect whether an AABB was outside a frustum plane and if so, the object which the AABB approximates was not rendered.

In this section we will cover a more generic implementation of a routine that will classify an AABB with respect to a plane and return a member of our CCollision::CLASSIFYTYPE enumeration (i.e., CLASSIFY_INFRONT, CLASSIFY_BEHIND or CLASSIFY_SPANNING). We will require a function such as this during the tree building phase where we wish to test the AABB of a detail area against the node split planes to determine which children it should be assigned to.

Classifying an AABB with respect to a plane is a simple case of finding the two corner points of the AABB that would be closest to the plane and furthest from the plane ***if the AABB was in front of the plane***. We refer to these as the *near* and *far* points. We know for example that as the near point is the closest point to the plane in a scenario where the box is in front of the plane, if this point is in front of the plane, then the whole AABB must be in the front space also. Furthermore, the far point describes the point on the box that would be farthest from the plane in the scenario (where the box was in front of the plane). If this point is behind the plane, the whole box must be behind the plane also. Finally, if the near and far points are on opposing sides of the plane, the box is spanning the plane.

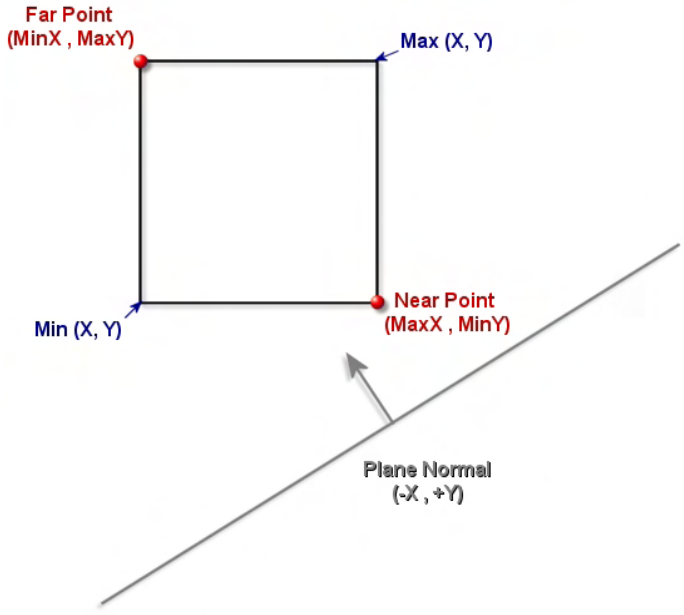


Figure 14.67

Calculating the near and far point is a simple case of analyzing the plane normal and picking the components from the AABB extents vectors to construct the point based on that normal. For example, if the x component of the normal is negative (the normal is facing in the direction of the negative X axis) then the x component of the near point will be the x component of the AABB max extents vector. If the normal is facing in the positive x direction, the x component of the AABB min extents vector describes the face of the cube that would intersect the front of the plane first were it in front of that plane. We do the same for each component of the normal. That is, for each component of the normal, if it is negative we use the maximum extents vector component for the near point and the minimum extents vector component for the far point. After doing this for each normal component, we will have constructed our two required points. In Figure 14.67 we demonstrate how this works.

Figure 14.67 depicts an AABB which is clearly in front of the plane. We also see the near and far points that have been generated based on the plane normal. Min and Max are the extent vectors of the AABB.

We can see that in this situation, if we were to move the box towards the plane, the near point would intersect the plane first and the far point would intersect the plane last. Let us examine how we generated the near and far points in this example.

The plane normal faces down the negative X axis and the Y component of the normal is positive (it is pointing up). We will keep things two dimensional for ease of illustration. Because the x component of the normal is negative, it means we use the x component from the max extents vector for the near point's x component (MaxX). Because the y component of the normal is positive, we use the y component of the AABB's minimum extents vector as the component for the near point. This gives us the point at the bottom right corner of the box shown in Figure 14.67. This is the point that would be closest to the plane if the box is in front of the plane. We can see that the reverse logic is true for the far point. If a normal component is negative, then the component from the far point is taken from the minimum extents vector of the box, otherwise it is taken from the maximum extents vector. As we can see in Figure 14.67, because the normal has a negative x component, we use MinX for its x component. Because the y component of the plane normal is positive, we take the y component from MaxY. This gives us the far point shown. Because the near point is in front of the plane, the entire box must be in front of the plane also, so we can return CLASSIFY_INFRONT.

It is important to realize that the near and far points are always calculated in this way regardless of the orientation of the plane or the position of the box. Therefore, the near point is not always the point that is nearest to the plane; it is the point that *would* be nearest to the plane *if the box was in front of the plane*.

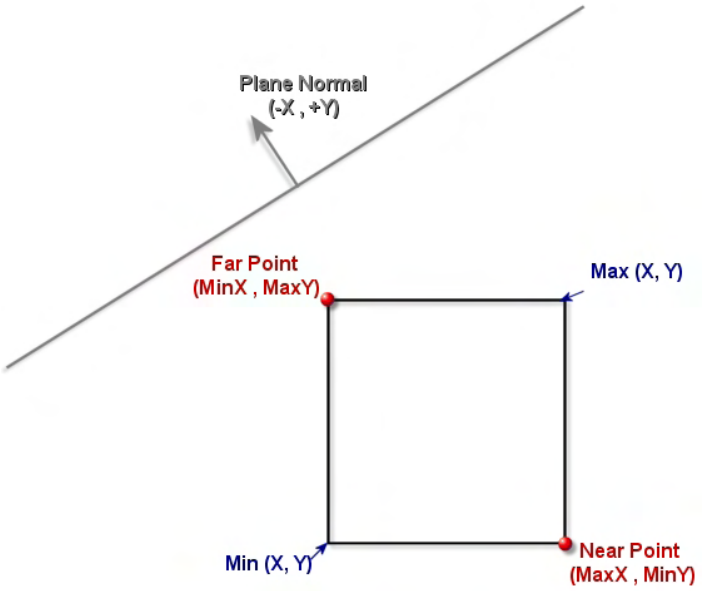


Figure 14.68

In Figure 14.68 we show the same plane being used but we have moved the box so that it is now behind the plane. Since the plane normal is the same, the near and far points of the box are calculated in exactly the same way. However, notice that because the box is behind the plane, the near point is no longer the point that is closest to the plane. We can see in this example that as the far point would be the last position on the cube to pass the plane if the cube was being moved from the planes front halfspace to its back halfspace, if the far point is behind the plane, the entire AABB must be behind the plane also and we can return CLASSIFY_BEHIND.

Of course, it is also clear to see when looking at Figures 14.67 and 14.68 that if, after calculating the near and far points, we find that they are on different sides of the plane, the box must be spanning the plane and we can return CLASSIFY_SPANNING.

The spanning case is illustrated in Figure 14.69. It demonstrates the fact that because we are now using a different plane normal, the near and far points are calculated completely differently. In this example,

because the plane normal has a negative x and y component, the near point is simply the AABB's minimum extents vector and the far point is the AABB's maximum extents vector.

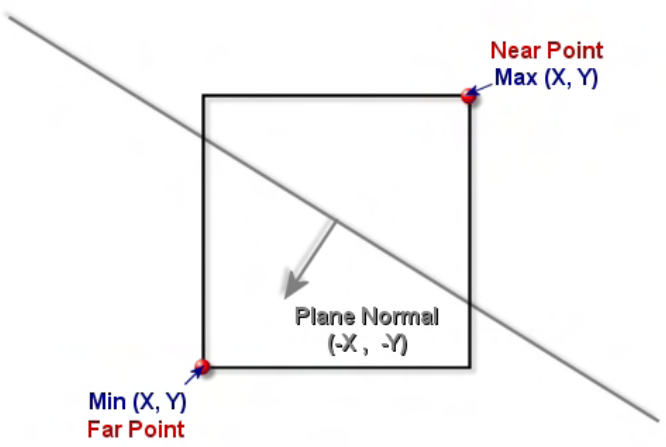


Figure 14.69

Below we see the code to the CCollision::AABBClassifyPlane function which implements the steps we have just discussed. Its parameters are the extent vectors of the AABB and the plane. The plane is passed in using ABCD format (a normal, ABC, and a distance from the plane to the origin of the coordinate system, D).

After calculating the near and far points, we classify the near point against the plane. If it is in front of the plane we can return CLASSIFY_INFRONT immediately since the whole box must be in front of the plane as well. If not, it means the near point is behind the plane so we test the far point. If the far point is in front of the plane, it means the box must be spanning the plane because the near and far points are in different half spaces. If this is the case, we return CLASSIFY_SPANNING. If not, the box must be behind the plane, so we can return CLASSIFY_BEHIND.

```
CCollision::CLASSIFYTYPE CCollision::AABBClassifyPlane( const D3DXVECTOR3& Min,
                                                        const D3DXVECTOR3& Max,
                                                  const D3DXVECTOR3& PlaneNormal,
                                                        float PlaneDistance )
{
    D3DXVECTOR3 NearPoint, FarPoint;

    // Calculate near / far extreme points
    if ( PlaneNormal.x > 0.0f ) { FarPoint.x  = Max.x; NearPoint.x = Min.x; }
    else                        { FarPoint.x  = Min.x; NearPoint.x = Max.x; }

    if ( PlaneNormal.y > 0.0f ) { FarPoint.y  = Max.y; NearPoint.y = Min.y; }
    else                        { FarPoint.y  = Min.y; NearPoint.y = Max.y; }

    if ( PlaneNormal.z > 0.0f ) { FarPoint.z  = Max.z; NearPoint.z = Min.z; }
    else                        { FarPoint.z  = Min.z; NearPoint.z = Max.z; }

    // If near extreme point is outside, then the AABB is totally outside the plane
    if ( D3DXVec3Dot( &PlaneNormal, &NearPoint ) + PlaneDistance > 0.0f )
         return CLASSIFY_INFRONT;
```

```
    // If far extreme point is outside, then the AABB is intersecting the plane
    if ( D3DXVec3Dot( &PlaneNormal, &FarPoint ) + PlaneDistance > 0.0f )
         return CLASSIFY_SPANNING;

    // We're behind
    return CLASSIFY_BEHIND;
}
```

Yet another function has now been added to our intersection library that we will find ourselves using time and time again as we progress with our studies. This concludes our discussion of the new intersection and classification routines we need to add to our library in order to build spatial hierarchies.

14.16 T-Junctions

Earlier in the lesson, we learned that any extensive clipping procedure will likely introduce T-junctions in the geometry. In this section we will examine what T-junctions are, the displeasing visual artifacts they exhibit, and a way to fix them. Since all of our trees will have the ability to produce clipped trees, all of our derived tree classes will want to have their T-junctions fixed after the build process has completed (if the tree is intended to be used for rendering). Since fixing T-junctions is exactly the same algorithm regardless of the tree type being used (it is just a repair pass on the polygons in the tree's polygon list), we have decided to place the T-junction removal code inside the Repair method of CBaseTree. This way, the same function can be used to fix the geometry built by any of our derived tree classes.

As T-junctions exhibit themselves only when the data that contains the T-junctions is being rendered, the derived class (or the application) will not have to issue the Repair call after the tree has been built as it will happen automatically. That is, the CBaseTree::PostBuild method, which is called from the derived class Build function after the tree has been built, will issue a call to the CBaseTree::BuildRenderData method. Although this method will be discussed in the following lesson, it essentially prepares the tree polygon data so that it is ready to be rendered. This includes making a call to the CBaseTree::Repair function. Of course, the BuildRenderData method returns immediately if the tree has not been given a valid 3D device pointer which means the CBaseTree::Repair method will only be called automatically for trees that you intend to render. If you are not using the tree for rendering (just collision queries) and you have not passed a 3D device pointer into the tree's constructor (avoiding the render data being built), but would still like to have the T-junctions repaired, your application can issue a call to the Repair method after the Build method has returned. Once again, remember that if a device pointer was passed to the constructor, the render data would have been built and the T-junction repair function called automatically.

14.16.1 What are T-Junctions?

A T-Junction is the meeting of geometry along a shared edge where the vertices along that shared edge are not equal in each object. This can happen a lot when clipping is being performed. A quad for example might be split into two pieces vertically at one node, and then one of those splits later gets split into two pieces horizontally at another node (see Figure 14.70). We can see in this example that the polygons (which may have originally been one quad before clipping was performed) all share a common edge. That is, the right edge of the left polygon would be butted up against the left edges of the two smaller polygons to its right. This is a classic T-junction scenario.

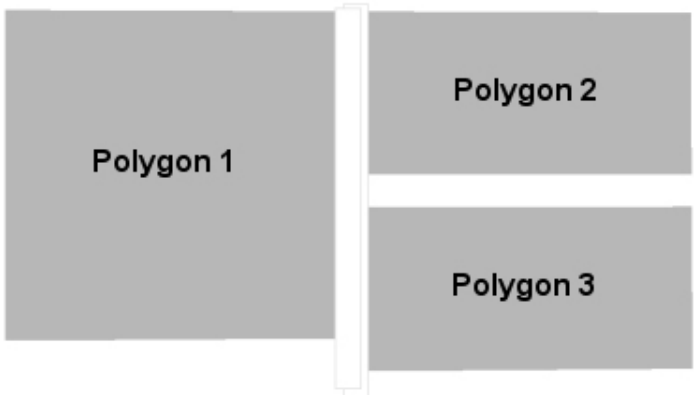


Figure 14.70

In this image we have deliberately separated the three neighboring polygons so that you can clearly see the polygon boundaries, but you should be able to recognize that if we were to slot all three polygons together so that there were no longer any gaps between them, they would look like a single large quad. Perhaps these three polygons were a single quad to begin with, but clipping broke it into three pieces. Nevertheless, we still want it to look like a single quad when rendered.

Of course, this same situation may arise simply because the artist has designed the geometry that way instead of it having been introduced by a clipping process. Either way, we have ourselves a T-junction to deal with and our scene may contain many of them.

While looking at the above image, if you tilt your head to the left, you should be able to see that the white gaps between the polygons form the letter ‘T’, which is where such a geometric configuration gets its name. If these polygons are neighbors (i.e., if their edges are touching) we would certainly have a problem that would cause lighting anomalies and other visual artifacts. In Figure 14.71 we have connected the polygons together to better show the problem. The polygon boundaries causing the T-junction are highlighted with red dashed lines.

Recall that a vertex lighting system using gouraud shading records lighting samples at the vertices for interpolation over the surface. We can see that while Polygon 1 shares its right hand edge with both Polygons 2 and 3, Polygons 2 and 3 have a vertex (shown as the red sphere) in the center of polygon 1’s right hand edge. But Polygon 1 does not have a vertex in that position since its vertices are the four corner points of the quad.

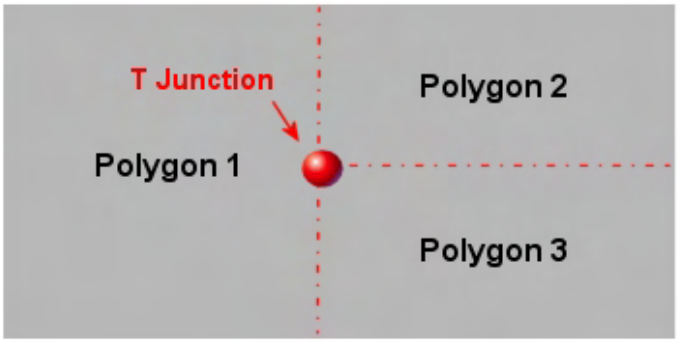


Figure 14.71

The problem occurs when a light source affects the center point of this construct where these three polygons meet. If we used a tightly focused beam (such as a small spot light for example), we would fully expect the influence of the light to affect all three polygons in the same way. However, as Figure 14.72 clearly shows, if we shine a small spot light at the center point of this construct, the bottom left vertex of Polygon 2 and the top left vertex of Polygon 3 are very close to the light source and as such, a high intensity light sample is recorded at those vertices. If Polygon 1 had a vertex in the same position, then it too would receive that same light sample which would be interpolated over its entire surface. The problem is that it does not have a vertex located there. In fact, Polygon 1's vertices are some distance from the light source. So although this construct of three polygons might seem to the user to be a large single wall polygon (for example), we have the lighting suddenly cut off as the light passes into Polygon 1's area because it does not have a vertex in a correct position to sample the light in the same way as its neighbors. This causes the lighting discontinuity illustrated in Figure 14.72.

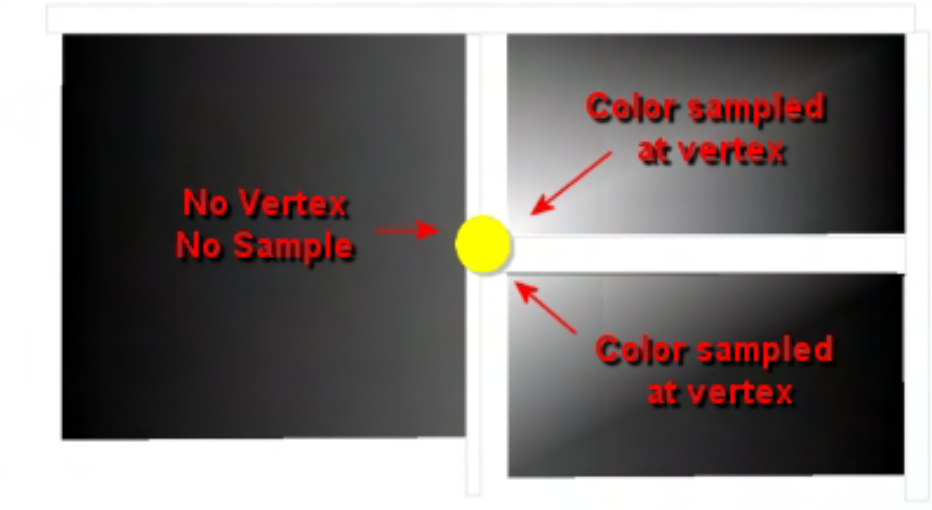


Figure 14.72

In Figure 14.73 we see the result is made much worse if the polygons are connected as they are supposed to be in this example. This is an obvious lighting discontinuity that would stick out like a sore thumb to even the most casual gamer.

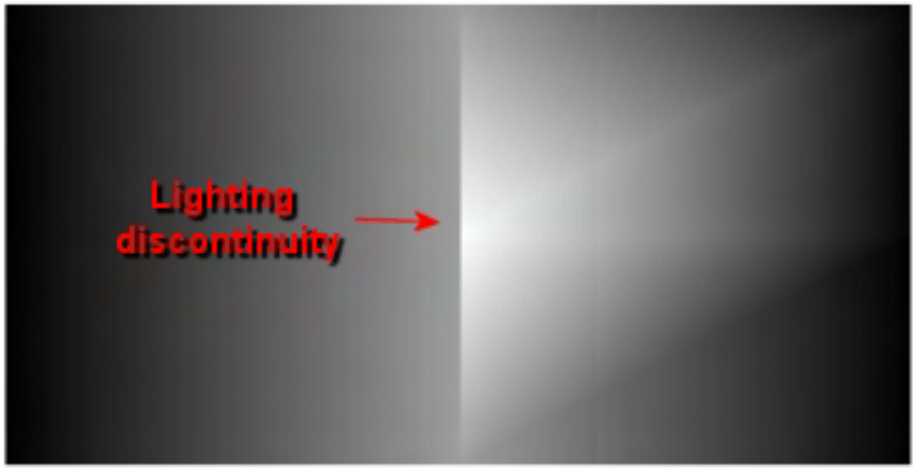


Figure 14.73

Whenever a T-junction occurs within a game level, we have the potential for a vertex lighting system to suffer these visual glitches. In fact, T-junctions are not just undesirable for a vertex lighting system; they also cause visual anomalies during rasterization. Many of you may have seen T-junctions in commercial games that did not even use a vertex lighting system. They can manifest themselves in the rasterization phase as a phenomenon called “sparklies”. Sparklies are sub-pixel gabs that appear when polygons that share edges with T-junctions are being rendered. In a T-junction scenario, rounding errors caused during rasterization occur differently along the edge that is missing the vertex versus the shared edges that have the additional vertex. The polygons appear to come apart at the shared edge ever so slightly at certain pixels, causing a gap. The difference in floating point rounding errors between the edges that do and do not contain the vertex causes a situation where neither edge renders a pixel into the frame buffer, thus allowing the background to show through. These gaps are often seen as a sparkle effect as lots of little oddly colored pixels shimmer in and out of existence on the display. These colored pixels are actually the colors of more distant polygons or the color that the frame buffer was initially cleared to before the frame was rendered.

In Figure 14.74 we see a screenshot from Lab Project 14.1 with the T-junction repair step turned off. Although it is hard to take a good screenshot of sparklies (they look much worse when the camera is actually moving), even in this image we can see some cyan colored pixels (the color the frame buffer was cleared to before the scene was rendered) showing through sub-pixel gaps on the floor.

Figure 14.74

When the player is actually walking along the floor and the level is fully animated, many of these little blue dots shimmer in and out of existence. This is very distracting to the player to say the least. In fact, it is for this reason, more so than the lighting discontinuities, that we must fix these T-junctions before the scene is rendered. Since we will be moving away from vertex lighting systems in Module III of this series and begin to favor more advanced techniques, T-junctions will no longer affect our lighting system. However, these sub-pixel gaps that occur during the rasterization of shared edges that contain T-junctions are clearly unacceptable.

As mentioned, T-junctions are not just caused by clipping geometry. They can be very easily introduced by the artist during level design. T-junctions can even exist between neighboring objects/meshes that will cause lighting anomalies (see Figure 14.75).

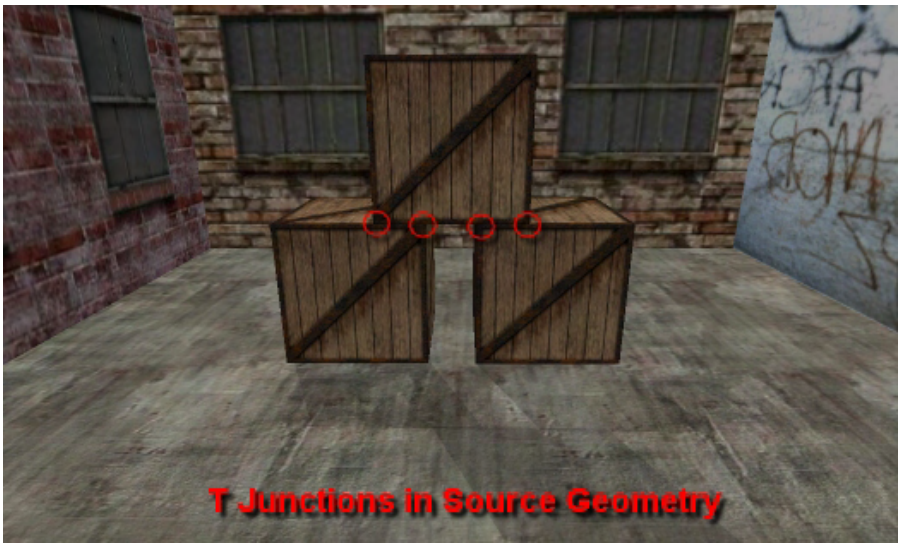


Figure 14.75

In Figure 14.75 we see how the project artist might innocently place three crates on top of each other in a staggered fashion. Because these are separate meshes, we will not have the sparklies problem, but this would still cause lighting discontinuities if a vertex lighting system was being employed.

Assuming the front face of each create is a single quad with the vertices in its four corners, we can see that top right vertex of the bottom left crate and the top left vertex of the bottom right crate partially share an edge with the bottom edge of the top crate.

Because the crates are staggered, the vertices in the bottom two crates that share the edge with the top crate have vertices in different positions versus those of the bottom edge of the top crate (and vice versa). We can also see that the bottom left and right vertices of the top crates are positioned in the middle of the top edges of the bottom crates and as such, we have four vertices that share an edge which do not each belong to both the polygons that share the edge. Take a look at Figure 14.76 and see how the vertex lighting system would light these crates if two tightly focused spot lights were set to illuminate the top left vertex of the bottom right crate and the top right vertex of the bottom left crate.

Figure 14.76

As you can see, the tightly focused beam illuminates only the points in space where the top right vertex of the bottom left crate and the top left vertex of the bottom right crate are located. The lighting color is sampled at this point and interpolated over the surfaces of the bottom two crates. However, the bottom edge of the top crate does not have vertices in these locales and its vertices (the bottom left and bottom right vertices of its quad) fall just outside the influence of the light source. As a result, the topmost crate has no vertices located in positions that receive light contributions and the create remains unlit. This has caused a very unnatural lighting result.

14.16.2 Fixing T-Junctions

Fixing T-junctions in our level is actually much simpler than you might assume. We essentially need to test the edges of each polygon in our scene, with the edges of every other polygon in the scene, searching for vertices in the edge of one polygon that exist along the edge of the first polygon. If such a vertex is found, we make a copy of the vertex and insert it into the polygon edge that has missing the vertex. Essentially, we are just finding any vertices in any polygon that may live in the middle of an edge and if found, we insert that vertex into the edge. The insertion of this extra vertex introduces another triangle primitive that will be needed to render the N-gon, so our primitive count will have grown after the T-junction repair process is complete. The basic pseudo code (lacking any optimizations) is:

```
For Each Polygon (PolygonA)
        For Each Edge (EdgeA)
                For Each Polygon (PolygonB)
                        For Each Vertex (VertexB)

                                D= Calc Distance from VertexB to EdgeA
                                ( If D !=0 [with epsilon] ) continue;

                                If  (VertexB is not between EdgeA.vertex1 and EdgeA.vertex2) continue;

                                // Vertex B is causing T junc with this EdgeA so inside it into polygon
                                PolygonA.InsertVertex( just after Edge1.vertex1);

                        End For Each Vertex
                End For Each Polygon
        End For Each Edge
End For Each Polygon
```

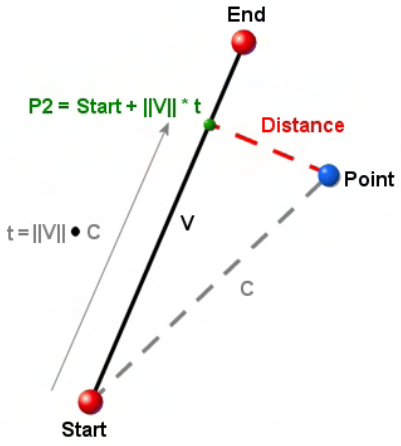
As you can see, we process each polygon and its edges one at a time. For each edge we test the vertices of every other polygon in the scene to see if that vertex causes a T-junction along the current edge. If it does, then we need to insert that vertex into the current edge so that both polygons have vertices in the same places, thus repairing the T junction. The key here is finding out whether the vertex we are currently testing is on the current edge we are testing because if it is not, it cannot possibly cause the T-junction with the current edge and we can skip it.

You can see in the above pseudo-code that for each vertex we calculate D, the shortest distance from the vertex to the infinite line on which the edge lives. If this is not zero (with tolerance) then the vertex cannot be on the edge. Obviously, if a point is on an edge, the distance from that point to the edge would be zero. If it is zero then we know that the vertex lies on the infinite line on which the edge/ray exists, but we do not know whether it lies within the line segment defined by the edge's two vertices. If it does not, then the vertex is not sharing an edge and we can ignore it. If it is between the two vertex positions then we definitely have found a vertex from another polygon that is on the edge we are currently testing in a position where the current edge does not have a vertex. When this is the case, we repair the T-junction by inserting this vertex into the edge just after the edge start vertex. This creates a new edge and a new triangle in the polygon and fixes the problem. Of course, when we insert this new vertex into the

polygon edge we must also generate its normal and texture coordinate. We can do this using interpolation based on its position between the two original vertices of the edge. Notice that testing Polygon A against Polygon B is not enough, we must also test Polygon B against Polygon A performing the same logic. The above algorithm will make sure this happens. The loops ensure that every polygon will be tested against every other polygon, so we will do ultimately perform n^2 tests (where n is the number of polygons).

It should be clear looking at the above code that the heart of this process is simply determining whether the vertex of a polygon lies on the edge of another. This process is reliant on us being able to calculate the distance from a vertex to a line segment (the edge). So we will need a function that will not only tell us the distance from a vertex to an infinite line, but will also inform us whether the point is inside the vertices of the line segment (the edge) assuming this distance is zero. Only if the distance from the vertex to the line is zero and if the point is positioned between the two edge vertices (contained within the line segment) do we wish our function to return a valid distance. So let us have a look at how we might calculate the distance from a point to a line segment.

14.16.3 Distance from a Point to a Line Segment



Calculating the Distance
from a Point to a Line Segment/Ray

Figure 14.77

Figure 14.77 illustrates the problem we are trying to solve. The blue sphere labeled Point is an arbitrary point that we would like to calculate the distance for. The line segment (the polygon edge in our example) has its two vertices shown as the red spheres labeled Start and End. The length of the red dashed line labeled Distance is what we are trying to find. If the distance is zero, then the point must lay on the same line as the edge. It may or may not be between the two vertices, but we will worry about that in a moment. For now, we are just interested in calculating the distance from the point to the infinite line on which the edge is contained.

What we are actually trying to do is calculate the position of the small green sphere in the diagram. This vector describes the projection of the point onto the line and thus, by subtracting this position from the original position, we get a vector matching the red dashed line in the diagram. The length of this vector is the distance we are seeking.

Finding the position of the green sphere is easy given the projection properties of the dot product. We first subtract the start of the line segment (the first vertex in the edge) from the point in question. This gives us vector C in the diagram. The length of this vector would tell us the distance from the origin of the ray/line segment to the point. Vector V in the diagram is calculated by the subtracting the edge start from the edge end position. V is essentially the ray delta vector with the start of the edge as the ray origin. If we normalize vector V, we can perform the dot product between unit length V and vector C to get *t*, the distance to travel in the direction of V from the ray start point to reach the green sphere.

All we have done is projected the length of vector C along the direction of V, which we know scaled it by the cosine of the angle between them. *t* now tells us how far we would have to travel along V from the start point to reach the green sphere. Therefore, we can calculate the position of this sphere simply by scaling unit length V by *t* and adding the resulting scaled V vector to the start of the edge. Now we have the position vector of the green sphere which we subtract from the position vector of the original point, which gives us the vector shown as the red dashed line in the diagram. We can then just return the length of this vector.

However, we also want our function to let the caller know when the point does exist on the infinite line for the edge, but is outside the boundaries of the edge vertices. That is no problem because we will know this as soon as we calculate *t*. Since V is unit length at this point, we know that *t* will be equal to zero at the edge start point and the pre-normalized length of V at the edge end point. Thus, as soon as we calculate *t*, we will return the highest possible distance value (FLT_MAX) if *t* is smaller than zero or greater than the original length of the edge. As our T-junction code will only take any action if the distance returned is zero, the function will only return zero if the point is indeed on the edge (and not just the infinite line on which the edge lies).

Next we see the code to the function. Because this is not a collision function, and would not really fit well in our CCollision namespace, we decided to add such a math utility function to a new cpp file called MathUtility.cpp. As we progress with our studies, we will place any functions we write which are general math utilities in this file. This function is contained in the MathUtility namespace defined in MathUtility.h.

MathUtility.h

```
namespace MathUtility
{
    float DistanceToLineSegment (   const D3DXVECTOR3& vecPoint,
                                    const D3DXVECTOR3& vecStart,
                                    const D3DXVECTOR3& vecEnd );
};
```

MathUtility.cpp – DistanceToLineSegment function code

```
float MathUtility::DistanceToLineSegment(       const D3DXVECTOR3& vecPoint,
                                                const D3DXVECTOR3& vecStart,
                                                const D3DXVECTOR3& vecEnd )
{
    D3DXVECTOR3 c, v;
    float       d, t;

    c = vecPoint  - vecStart;
    v = vecEnd - vecStart;
    d = D3DXVec3Length( &v );

    // Normalize V
    v /= d;

    // Calculate final t value
    t = D3DXVec3Dot( &v, &c );

    // Check to see if 't' is beyond the extents of the line segment
```

```
    if (t < 0.01f)     return FLT_MAX;
    if (t > d - 0.01f) return FLT_MAX;

    // Calculate intersection point on the line
    v.x = vecStart.x + (v.x * t);
    v.y = vecStart.y + (v.y * t);
    v.z = vecStart.z + (v.z * t);

    // Return the length
    return D3DXVec3Length( &(vecPoint - v) );
}
```

The function above mirrors Figure 14.77. It is passed the two position vectors of the line segment (the start and end points) and the point we wish to classify. We first calculate c and v as discussed, then store the length of the line segment (edge) in d. We will need this value later and we are just about to normalize it. We then normalize vector V by dividing it by its length and calculate *t* by dotting unit length vector v with c. This projects the length of c onto v (scaled by the cosine between them), returning a *t* value in the range [0, d] for points that are contained between the two end points.

If *t* is less than zero or larger than the length of the line segment, then it does not matter what the distance to the infinite line is; this point could not possibly be contained in the line segment. Its projection places it on the infinite line either prior to the start vertex or after the end vertex. When this is the case we return a distance of FLT_MAX (the maximum value that can be stored in a float). If this is not the case, then we know that the projection of the point onto the line segment did indeed produce a point (the green sphere in Figure 14.77) that is on the edge. However, we have no idea how far it is from that edge; only if it is at a distance of zero is the point truly on the edge.

Next we calculate the position of the green sphere in Figure 14.77 and store the result in v. We calculate it by scaling v by *t* and adding the result to the start vector of the edge. Now that we have the projected position stored in v, we can simply subtract it from the position of the original point and return the length of the resulting vector. If this length is zero, the point is on the edge and we have ourselves a T-junction that needs repair.

14.16.4 The T-Junction Repair Code

We now have everything we need to write our T-junction repair code. However, when we examine the pseudo-code discussed previously, we realize immediately that it will be an incredibly slow operation. In that vanilla description of the algorithm, we had to test each edge of every polygon against every vertex in every other polygon. Since most of us will be reluctant to wait for an extended amount of time for our T-junctions to be repaired, we need to speed this up as much as possible.

As it happens, by the time the repair process is invoked, the spatial tree will have already been built. Therefore we can reduce the time it takes to compute this process down to a mere fraction of the time it would otherwise take if we incorporate a spatial hierarchy into the process. At the point the repair process is called, each polygon will have already been assigned to its leaves and each polygon will have

a bounding box of its own. Therefore, much like we will implement the broad phase collision step, we will only have to test a mere handful of vertices during the repair process for each polygon's edge.

For each polygon, we will call the CollectLeavesAABB function which will be discussed in detail shortly. This is the function we will implement in our derived classes that will traverse the tree and return any leaves whose AABB intersects the query AABB. In this instance, the query AABB will be the AABB of the polygon we are currently repairing. The CollectLeavesAABB function will return a list of all leaves that intersect the polygon's AABB, allowing us to reject virtually every other polygon in the scene from consideration. Only the polygons in the returned leaves will need to be tested against each other. These are the polygons that are in close proximity to the polygon currently being repaired and might possibly share an edge with it. We will then loop through each polygon in the returned leaves and compare the bounding box of the polygon with the bounding box of the polygon we are currently repairing. Only when the two bounding boxes intersect will we test each vertex in that polygon against each edge in the polygon currently being repaired.

Repair - CBaseTree

The CBaseTree::Repair function is actually a wrapper around the core T-junction repair process. It takes care of looping through each polygon and collecting the leaves that intersect its bounding box. It also takes care of comparing the AABBs of the two polygons to determine whether a T-junction repair test should be done on them. Once it finds two polygons that have intersecting AABBs (and thus the potential to share an edge), it calls the CBaseTree::RepairTJunctions function to actually repair it. We will look at this method after we have examined the code to the main Repair function. This code is an excellent example of how hierarchies can speed up virtually any task done at the scene level.

In the first section of the code we set up a loop to iterate through all the CPolygon pointers stored in the tree's m_Polygons linked list. Remember, this list will contain every polygon being used by the tree. For each polygon in the list we will then extract its AABB. This is our current polygon being tested for repair.

```
bool CBaseTree::Repair( )
{
    ULONG                   SrcCounter, DestCounter, i;
    CPolygon               *pCurrentPoly, *pTestPoly;
    PolygonList::iterator   SrcIterator, DestIterator;
    LeafList::iterator      LeafIterator;
    LeafList                LeafList;

    // No-Op ?
    if (m_Polygons.size() == 0 ) return true;

    try
    {
        // Loop through Faces
        for ( SrcCounter = 0,
              SrcIterator = m_Polygons.begin();
              SrcIterator != m_Polygons.end();
              ++SrcCounter,
```

```
              ++SrcIterator )
        {
            // Get poly pointer and bounds references for easy access
            pCurrentPoly = *SrcIterator;
            if ( !pCurrentPoly ) continue;

            D3DXVECTOR3 & SrcMin = pCurrentPoly->m_vecBoundsMin;
            D3DXVECTOR3 & SrcMax = pCurrentPoly->m_vecBoundsMax;
```

At the head of the function we declared a local variable LeafList, which is a variable of an STL list that stores ILeaf pointers. We make sure this list is empty and then send this list along with the AABB of the current polygon into the CollectLeavesAABB method.

```
            // Retrieve the list of leaves intersected by this poly
            LeafList.clear();
            CollectLeavesAABB( LeafList, SrcMin, SrcMax );
```

The CollectLeavesAABB code will be discussed in a moment since it is implemented in the derived classes. The method fills the passed leaf list with any leaves in the tree which intersect the bounding box of the polygon. On function return, LeafList will contain a list of ILeaf pointers which contain polygons that have the potential to cause T-junctions with the current polygon (i.e., a broad phase).

We will now loop through each leaf in the returned list and extract its pointer so that we can access its data. Notice how LeafList is defined to store ILeaf pointers, but we know that they will contain pointers to CBaseLeaf objects because this is the type of leaf we will allocate and store during the build process. Therefore, we cast the current ILeaf pointer to a CBaseLeaf.

```
            // Loop through each leaf in the set
            for ( LeafIterator = LeafList.begin();
                  LeafIterator != LeafList.end(); ++LeafIterator )
            {
                CBaseLeaf * pLeaf = (CBaseLeaf*)(*LeafIterator);
                if ( !pLeaf ) continue;
```

For the current leaf, we will set up a loop that allows us to retrieve each of its polygon pointers one at a time and perform an AABB/AABB test between it and the current polygon being repaired. Notice how we use our new CCollision::AABBIntersectAABB function to perform the AABB test between the two polygons. If it returns true, the polygon do have intersecting bounding volumes and we will have to feed them both into CBaseTree::RepairTJunctions where they will be tested at the edge/vertex level. The remaining code of the Repair method is shown below.

```
                // Loop through Faces
                DestCounter = pLeaf->GetPolygonCount();

                for ( i = 0; i < DestCounter; ++i )
                {
                    // Get poly pointer and bounds references for easy access
                    pTestPoly = pLeaf->GetPolygon( i );
                    if ( !pTestPoly || pTestPoly == pCurrentPoly ) continue;
                    D3DXVECTOR3 & DestMin = pTestPoly->m_vecBoundsMin;
                    D3DXVECTOR3 & DestMax = pTestPoly->m_vecBoundsMax;
```

```
                    // Do polys intersect
                    if ( !CCollision::AABBIntersectAABB( SrcMin,
                                                         SrcMax,
                                                         DestMin,
                                                         DestMax ) ) continue;

                    // Repair against the testing poly
                    RepairTJunctions( pCurrentPoly, pTestPoly );


                } // Next Test Face

            } // Next Leaf

        } // Next Current Face

    } // End Try Block

    catch ( ... )
    {
        // Clean up and return (failure)
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

So we have seen that While the CBaseTree::Repair method is called post-build to repair T-junctions, the actual function code really just implements a broad phase testing process for the real T-junction repair code. Our collision system will use the hierarchy in a very similar way. The only polygons we have to test at the edge/vertex level are polygons that have intersecting bounding volumes. This will typically be a few polygons at most for each polygon being tested. That is certainly a lot more efficient than testing every single vertex and every single polygon against every edge of every other polygon in the scene.

RepairTJunctions - CBaseTree

The code to this function is quite small and straightforward. It is passed two polygon pointers. The first polygon is the polygon currently having its edges tested for T-junctions. It is the polygon that will have vertices added to it if a T-junction is identified. The second polygon is the polygon that is going to have each of its vertices tested against each of the edges of the first polygon to see if any of them share that edge and cause a T-junction.

The function begins by setting up a loop to iterate through every edge in the polygon. The loop will step through each vertex and the vertex ahead of it in the array for the edge that is currently being tested.

```
void CBaseTree::RepairTJunctions( CPolygon * pPoly1, CPolygon * pPoly2 )
```

```
{
    D3DXVECTOR3 Delta;
    float       Percent;
    ULONG       v1, v2, v1a;
    CVertex     Vert1, Vert2, Vert1a;

    // Validate Parameters
    if (!pPoly1 || !pPoly2) return;

    // For each edge of this face
    for ( v1 = 0; v1 < pPoly1->m_nVertexCount; v1++ )
    {
        // Retrieve the next edge vertex (wraps to 0)
        v2 = ((v1 + 1) % pPoly1->m_nVertexCount);

        // Store verts (Required because indices may change)
        Vert1 = pPoly1->m_pVertex[v1];
        Vert2 = pPoly1->m_pVertex[v2];
```

v1 is always the index of the current vertex and v2 is the index of the next one in the winding of the polygon. These two vertex indices form the current edge that is to be tested against. Notice that v2 is calculated using the modulus operator so that when v1 is equal to the last vertex in the winding, v2 will wrap around to vertex 0 (the final edge in a polygon is formed by vertex N and vertex 0, where N is the number of vertices in the polygon). In the above code you should notice that once these indices are calculated, we use them to fetch the actual vertex data from the CPolygon's vertex array.

We now have the two vertices of the edge we wish to test, so now we need to loop through each vertex in the second polygon and calculate its distance from the edge. If the DistanceToLineSegment method that we just wrote is passed the two vertices of the edge and the vertex from polygon 2, and it returns zero (with tolerance), the vertex of the second polygon causes a T-junction with the edge and steps must be taken to repair it.

```
        // Now loop through each vertex in the test face
        for ( v1a = 0; v1a < pPoly2->m_nVertexCount; v1a++ )
        {
            // Store test point for easy access
            Vert1a = pPoly2->m_pVertex[v1a];

            // Test if this vertex is close to the test edge
            if ( MathUtility::DistanceToLineSegment(  (D3DXVECTOR3&)Vert1a,
                                                      (D3DXVECTOR3&)Vert1,
                                                      (D3DXVECTOR3&)Vert2 )<0.01f )
            {
```

In this next code block we repair the T-junction. This is done by simply inserting a new vertex in the polygon immediately after the first vertex in the edge we are testing (v1). That is, we must insert it at the position in the vertex array currently occupied by the second vertex v2. This will nudge all vertices in the winding order up by one position in the array, creating the current edge between v1 and the new vertex we have just inserted. It also adds a new edge to the polygon between the new vertex and what used to be the second vertex in the current edge.

The following code demonstrates inserting a new vertex into the polygon and setting its position equal to the vertex in the second polygon that caused the T-junction.

```
                // Insert a new vertex within this edge
                long NewVert = pPoly1->InsertVertex( (USHORT)v2 );
                if (NewVert < 0) throw std::bad_alloc();

                // Set the vertex pos
                CVertex * pNewVert = &pPoly1->m_pVertex[ NewVert ];

                pNewVert->x = Vert1a.x;
                pNewVert->y = Vert1a.y;
                pNewVert->z = Vert1a.z;
```

Because our polygon has a vertex in exactly the same position as the one that caused the T-junction in the second polygon, we have repaired it. However, we also have to generate the texture coordinate information and normal for this newly inserted vertex.

Just as we did in the polygon splitting routine, we create the new normal and texture coordinates for the vertex by interpolating the texture coordinates and normal vectors stored in the original edge vertices based on the distance along that edge where we have inserted the new vertex. In the following code we divide the length of the new edge (vert1 to new vertex) by the length of the original edge (Vert1 to Vert2) to get a *t* value back that describes the position of the new vertex parametrically along the original edge. We use this to weight the interpolation as we have done so many times before. Below we see the remainder of the function.

```
                // Calculate the percentage for interpolation calcs
                Percent = D3DXVec3Length( &(*(D3DXVECTOR3*)pNewVert –
                                            (D3DXVECTOR3&)Vert1) )
                          /
                          D3DXVec3Length( &((D3DXVECTOR3&)Vert2 –
                                           (D3DXVECTOR3&)Vert1) );

                // Interpolate texture coordinates
                Delta.x      = Vert2.tu - Vert1.tu;
                Delta.y      = Vert2.tv - Vert1.tv;
                pNewVert->tu = Vert1.tu + ( Delta.x * Percent );
                pNewVert->tv = Vert1.tv + ( Delta.y * Percent );

                // Interpolate normal
                Delta            = Vert2.Normal - Vert1.Normal;
                pNewVert->Normal = Vert1.Normal + (Delta * Percent);
                D3DXVec3Normalize( &pNewVert->Normal, &pNewVert->Normal );

                // Update the edge for which we are testing
                Vert2 = *pNewVert;

            } // End if on edge
        } // Next Vertex v1a
    } // Next Vertex  v1
}
```

Not only have we learned how to fix T-junctions, we have also finished our coverage of CBaseTree for this chapter. In the next chapter we will add a few more functions and variables to CBaseTree so that it offers an efficient rendering strategy. For now though, we will remain focused on tree building and using trees to perform collision queries.

With the base functionality in place, we can now finally look at the code to the derived classes. For the most part, these classes will contain only a few methods. This is due to the fact that, apart from the building and traversal functions, everything else has already been implemented in CBaseTree.

In the following sections of the textbook we will cover the source code to the quad-tree, the oct-tree and the kD-tree, one at a time and in that order. Most of their implementations will be identical, with only slightly different traversal and build logic. Thus, covering the building process for each tree should be a relatively painless affair after we have covered the build function for the first. Let us start by looking at our first derived class, CQuadTree. This is the class that implements the vanilla quad-tree and it is the first tree class we have developed so far that is designed to be instantiated by the application.

14.17 CQuadTree - Implementation

Contained in the source files CQuadTree.cpp and CQuadTree.h is the code for our vanilla quad-tree implementation. It is derived from CBaseTree as all tree classes will be and as such, there are several functions that we must implement because they are required by ISpatialTree but not implemented in CBaseTree. For example, we must implement the Build method. This method will recursively build a tree of quad-tree nodes using the polygon and detail area data that has been registered with the tree and populate the leaf lists with CBaseLeaf objects. After the Build function has returned, the quad-tree will be complete and we will have a tree of quad-tree nodes. Each node will contain a bounding box large enough to contain the polygon data and detail areas that are assigned to any of its child nodes. At each point during the build process, a node will be divided into four equal quadrants and each quadrant assigned to a child of that node.

Other functions that are specified in ISpatialTree but not implemented in CBaseTree are the collision functions, such as CollectLeavesAABB and CollectLeavesRay. These must also be implemented in this class to provide the traversal logic that steps through the nodes of a quad-tree searching for leaf nodes. These will be small and simple methods since the collision code and much of the core logic resides in CBaseTree and CCollision.

14.17.1 CQuadTree Node – The Source Code

Although CBaseTree implements the CBaseLeaf object that all trees will use to contain the polygon and detail area data at a leaf node, it does not implement a node object that will be used to link the nodes of the hierarchy together. This is because the node structure of each derived class will be different since they have different child counts. A quad-tree for example must store pointers to four child nodes, an oct-tree must store pointers to eight children, and a kD-tree will store only two child node pointers. This

means it is up to the programmer who implements the derived class to also define the node structure the tree will use. The structure we use for the quad-tree nodes is shown below.

```
class CQuadTreeNode
{
public:

    // Constructors & Destructors for This Class.
     CQuadTreeNode( );
    ~CQuadTreeNode( );

    // Public Functions for This Class
    void SetVisible( bool bVisible );

    //Public Variables for This Class
    CQuadTreeNode * Children[4];        // The four child nodes
    CBaseLeaf     * Leaf;               // If this is a leaf, store here.
    D3DXVECTOR3     BoundsMin;          // Minimum bounding box extents
    D3DXVECTOR3     BoundsMax;          // Maximum bounding box extents
    signed char     LastFrustumPlane;   // The frame-to-frame coherence plane
};
```

This very simple structure contains pointers to four children and a CBaseLeaf pointer. The only nodes that will not have NULL assigned to their leaf pointer will be terminal nodes (nodes at the ends of branches that have been made into leaf nodes). When traversing the tree searching for leaf nodes, we know that a leaf node is any node that does not have NULL assigned to its leaf pointer. If it is leaf node, the attached CBaseLeaf object will contain all the polygon data and detail area data assigned to that terminal node. Following the leaf pointer in the declaration are two vectors in which the node's bounding box will be stored and a fifth member that will be explained in detail in the next lesson.

Finally, notice that this class has a method called SetVisible which the tree can call to make all its children visible. A node does not hold a visible status but this function is a recursive function that traverses the children of the node and sets the visibility status of any leaves underneath it to true. This will be used in our visibility testing methods. For example, if a node's volume is completely contained inside the frustum, then all direct and indirect children of that node must also be inside it. When this is the case, we no longer have to perform frustum/AABB tests on each of its children since we know the entire branch starting at that node must be contained in the frustum. When this is the case, this method is called to immediately set the visibility status of all child leaves to true.

Constructor - CQuadTreeNode

The quad-tree node constructor makes sure that its leaf and child pointers are set to NULL and that the last frustum plane index is set to -1 (no plane).

```
CQuadTreeNode::CQuadTreeNode()
{
    unsigned long i;

    // Reset / Clear all required values
```

```
    Leaf             = NULL;
    LastFrustumPlane = -1;

    // Delete children
    for ( i = 0; i < 4; ++i ) Children[i] = NULL;
}
```

Destructor - CQuadTreeNode

The destructors for all of our derived node types will work the same way. When a node is destroyed, it deletes its child nodes, which causes their destructors to trigger, which then deletes their child nodes, and so on down the tree. This causes a cascade effect that removes every node of the tree from memory. Thus, the tree just has to delete the root node to kick off this process. The code is shown below.

```
CQuadTreeNode::~CQuadTreeNode()
{
    unsigned long i;

    // Delete children
    for ( i = 0; i < 4; ++i )
    {
        if ( Children[i] ) delete Children[i];
        Children[i] = NULL;

    } // Next Child

    // Note : We don't own the leaf, the tree does, so we don't delete it here

    // Reset / Clear all required values
    Leaf = NULL;
}
```

Note that if the node is a terminal node and has a leaf attached to it, we do not delete the leaf from memory, we simply set its pointer to NULL. The reason we do not worry about the node deleting its leaf is because the leaf pointers are also stored in the tree's m_Leaves STL list. The tree can delete all the leaves in one go afterwards simply by emptying this list.

Since the constructor and destructor for all of our various node types will be the same as this one (just with more or less children to delete) we will not show the code to the constructors and destructors of other node types from this point forwards.

SetVisible - CQuadTreeNode

Although we will not see the SetVisible method being used until we cover the rendering system of CBaseTree (in the next lesson), its code is very simple so we will cover it here. As you can see, it first tests to see if its leaf pointer is non NULL. If it is, then this is a leaf and we set the leaf's visible status to that of the boolean passed. This means this recursive function can be used to turn on or off the visibility

status of all leaves down a given branch of the tree (using the boolean parameter). If the current node is a leaf node then our job is done and we return. If the current node is not a leaf, then the function calls itself recursively for each of its children.

```
void CQuadTreeNode::SetVisible( bool bVisible )
{
    unsigned long i;

    // Set leaf property
    if ( Leaf  ) { Leaf->SetVisible( bVisible ); return; }

    // Recurs down if applicable
    for ( i = 0; i < 4; ++i )
    {
        if ( Children[i] ) Children[i]->SetVisible(bVisible);

    } // Next Child
}
```

14.17.2 CQuadTree – The Source Code

Below we see the CQuadTree class declaration (see CQuadTree.h) which shows the functions and members we added beyond those inherited from CBaseTree. In the following listing you can see that the first group of methods are those that must be implemented to support the ISpatialTree interface. For example, we know that every derived class must provide the implementation for the Build, CollectLeavesAABB, and the CollectLeavesRay methods since these are the application's only means to build and query the tree. Even methods in CBaseTree, such as the Repair method, rely on the CollectLeavesAABB method being implemented in the derived class so that polygon neighbors can be quickly determined.

Let us look at the public methods implemented to the support the base class first.

```
class CQuadTree : public CBaseTree
{
public:

    // Constructors & Destructors for This Class.
    virtual ~CQuadTree( );
             CQuadTree( LPDIRECT3DDEVICE9 pDevice, bool bHardwareTnL );

    // Public Virtual Functions for This Class (from base).
    virtual bool Build                  ( bool bAllowSplits = true );
    virtual void ProcessVisibility      ( CCamera & Camera );

    virtual bool CollectLeavesAABB      ( LeafList & List,
                                          const D3DXVECTOR3 & Min,
                                          const D3DXVECTOR3 & Max );

    virtual bool CollectLeavesRay       ( LeafList & List,
                                          const D3DXVECTOR3 & RayOrigin,
```

```
                                        const D3DXVECTOR3 & Velocity );
    virtual void DebugDraw              ( CCamera & Camera );

    virtual bool GetSceneBounds         ( D3DXVECTOR3 & Min, D3DXVECTOR3 & Max );
```

The ProcessVisibility method is specified in ISpatialTree and should be called by the application prior to calling the ISpatialTree::DrawSubset method. Although we will not discuss the rendering system until the next chapter, the purpose of this function is to traverse the quad-tree with the passed camera and set the visible status of any leaves according to whether they are currently inside or outside the frustum. The GetSceneBounds function should return the bounding box of the entire area compiled into the quad-tree (which is the bounding volume of the root node). Finally, the DebugDraw method does not have to be implemented in any derived class as it has a default implementation in ISpatialTree which does nothing. However, each of our derived classes will override this method so that we can traverse the tree and render their leaf node bounding boxes to visualize what we have created.

Below we see the private class methods. These are basically the methods that the public methods shown above use to accomplish their tasks. The functions listed above are really just doorways to the private recursive procedures. For example, the Build method creates the root node and builds the initial polygon list for that node and then calls the BuildTree function which recursively calls itself until the tree is complete. The same is true of the CollectLeavesAABB method which is really a one line function that makes sure the CollectAABBRecurse method is first called using the root node.

```
private:

    // Private Functions for This Class
    bool         BuildTree              ( CQuadTreeNode * pNode,
                                          PolygonList PolyList,
                                          DetailAreaList AreaList,
                                          const D3DXVECTOR3 & BoundsMin,
                                          const D3DXVECTOR3 & BoundsMax );

    void         UpdateTreeVisibility   ( CQuadTreeNode * pNode,
                                          CCamera & Camera,
                                          UCHAR FrustumBits = 0x0 );

    bool         DebugDrawRecurse       ( CQuadTreeNode * pNode,
                                          CCamera & Camera,
                                          bool bRenderInLeaf );

    bool         CollectAABBRecurse     ( CQuadTreeNode * pNode,
                                          LeafList & List,
                                          const D3DXVECTOR3 & Min,
                                          const D3DXVECTOR3 & Max
                                                bool           bAutoCollect = false
);

    bool         CollectRayRecurse      ( CQuadTreeNode * pNode,
                                          LeafList & List,
                                          const D3DXVECTOR3 & RayOrigin,
                                          const D3DXVECTOR3 & Velocity );
```

Because CBaseTree provides us with the leaf list, detail area list, and polygon list in which to store our data, we only need to add a few more members in this class. The first is a CQuadTreeNode pointer which will point to the root node of the tree once it has been built. Given our extensive exposure to hierarchies in this course, we know that we only need to store the root node to a tree/hierarchy for its methods to be able to perform a complete traversal of every node in the tree. We also add a boolean member called m_bAllowSplits which simply stores whether a clipped tree is being built. This is determined by the value of the boolean parameter passed into the build function, which is stored in this member. The remainder of the class declaration is shown below.

```
      //-----------------------------------------------------------------------
      // Private Variables for This Class
      //-----------------------------------------------------------------------
      CQuadTreeNode * m_pRootNode;      // The root node of the tree
      bool            m_bAllowSplits;   // Is splitting allowed?

      // Stop Codes
      float          m_fMinLeafSize;     // Min leaf size stop code
      ULONG          m_nMinPolyCount;    // Min polygon count stop code
      ULONG          m_nMinAreaCount;    // Min detail area count stop code
};
```

Notice that we now have three members at the very bottom which store a minimum leaf size, a minimum polygon count, and a minimum detail area count. The values of these members will be passed in as parameters to the constructor by the application to provide a degree of control over the way the tree is built. When the tree is being compiled, a decision must be made at each node as to whether the node satisfies the requirements to become a leaf node. We implement three stop codes to that can be set by the application to influence this decision. If any of these stop codes are reached, the current node will have no children generated for it and will be made a leaf node. The three stop codes we use for the quad-tree are defined in CQuadTree.h and are discussed next.

The first stop code is the size of the leaf. We currently set our default such that, if the bounding volume of a node is determined to have a diagonal length (e.g., bottom left to top right) of less than 300 world space units, we decide that this node's volume is so small that we do not wish to further subdivide and create any more children down that branch of the tree. When this is the case, the node's child pointers are set to NULL and a new CBaseLeaf object is created and attached to the node. The leaf is also added to the tree's leaf list and the leaf itself has the pointers for all polygons and detail areas that made it into that node added to its internal arrays.

The second stop code occurs during the creation of a node when we determine that less than a specified number of polygons have made it into that node. The default value for this is 600. This means that regardless of how large a node's volume might be, if it contains less than 600 polygons we decide that further subdividing this polygon set would be futile and the node is made a leaf, exactly as described in the previous paragraph.

The third stop code is one that allows us to stop subdividing a node's volume if less than a certain number of detail areas exist in that node. By default we set this to zero so that detail areas play no part in determining whether a node should be made a leaf node. However, if you were to set this value to 1 (for example) then whenever a node was encountered that had only one detail area in it and the polygon list was lower than the polygon count stop code, a leaf node will be created. This stop code may be a little

hard to understand, and most of the time it will go unused and be set to zero. However, it can become very useful when you are building a tree that has no polygon data and consists only of detail areas, since it allows us to control the size of the leaf nodes under those conditions. For example, consider what happens if we compiled our terrain into a quad-tree. As discussed previously in this lesson, we will not want to compile every terrain polygon into the tree since the terrain data is already organized into a series of render ready terrain blocks (sub-meshes). What we might decide to do instead is just build a tree of detail areas. We could for example create a bounding box around each terrain block and register each of these volumes as detail areas with the tree. When the tree is compiled (which contains no polygon data) the only stop code would be the leaf size. However, this might carve up the space into many more leaves than necessary such that a given terrain block's volume could be contained in many leaves. If all we are using the tree for is quickly querying which terrain block is visible, we would rather have each terrain block fit into a single leaf equal to its size. Ideally, if the terrain was divided into a 10x10 grid of terrain blocks, we would like that space divided up into a grid of 10x10 terrain block sized leaf nodes. In this situation we could set the detail area count stop code to 1, and as soon as a node had only one detail area assigned to it, it would be made into a leaf node containing that single detail object. This means our tree would contain a leaf node for each terrain block allowing us to keep the tree as shallow as possible and therefore speeding up tree traversals.

The Constructor - CQuadTree

The CQuadTree constructor accepts two required parameters and three optional parameters. The first is a pointer to a Direct3D device which the rendering system will use to render the tree. The second is a boolean describing whether this is a hardware or software device. This information will be needed by the rendering system when creating the index and vertex buffers that will store the geometry for the render system.

```
CQuadTree::CQuadTree( LPDIRECT3DDEVICE9 pDevice,
                      bool bHardwareTnL
                      float fMinLeafSize /* = 300.0f */,
                      ULONG nMinPolyCount /* = 600 */,
                      ULONG nMinAreaCount /* = 0 */ ) : CBaseTree( pDevice, bHardwareTnL )
{
    // Clear required variables
    m_pRootNode = NULL;

    // Store stop code values
    m_fMinLeafSize  = fMinLeafSize;
    m_nMinPolyCount = nMinPolyCount;
    m_nMinAreaCount = nMinAreaCount;
}
```

The device pointer and its boolean hardware status are immediately sent to the CBaseTree constructor where they are stored in CBaseTree member variables. These will be used after the tree is built to set up the render system. The final three parameters contain the stop codes that the application can pass to influence which nodes are candidates for becoming leaf nodes during a build. We simply store these values in their corresponding member variables.

In our coverage of CBaseTree we learned that the CBaseTree::PostBuild method was to be called after the derived class's Build method completed its task and built the tree (CBaseTree::PostBuild will be called from the CQuadTree::Build function just before it returns). The PostBuild method issues a call to the CBaseTree::BuildRenderData method which we will cover in the next lesson. This method will build the render data if NULL was not passed as the device pointer. Therefore, we can inform the tree that it will not be used for rendering by passing NULL as the first parameter and save the memory that would otherwise be allocated to prepare and contain the tree data in a renderable format.

Destructor – CQuadTree

At first glance, the CQuadTree destructor appears to neglect to free a lot of its memory. It simply deletes the root node, which triggers the deletion of every node in the tree.

```
CQuadTree::~CQuadTree()
{
    // Release any resources
    if ( m_pRootNode ) delete m_pRootNode;

    // Clear required variables
    m_pRootNode = NULL;
}
```

What about the leaves, the polygons and the detail area? Keep in mind that our classes all have virtual destructors, so the base class's destructor will automatically be called. CBaseClass actually manages the polygon, detail area, and leaf list data, so it is this destructor that releases them (shown below).

As you can see, the base class destructor first loops through the linked list of polygons used by the tree and deletes each one before emptying the pointer list and freeing up its memory.

```
CBaseTree::~CBaseTree()
{
    PolygonList::iterator       PolyIterator = m_Polygons.begin();
    LeafList::iterator          LeafIterator = m_Leaves.begin();
    DetailAreaList::iterator    AreaIterator = m_DetailAreas.begin();
    ULONG                       i;

    // Iterate through any stored polygons and destroy them
    for ( ; PolyIterator != m_Polygons.end(); ++PolyIterator )
    {
        CPolygon * pPoly = *PolyIterator;
        if ( pPoly ) delete pPoly;

    } // Next Polygon

    m_Polygons.clear();
```

We then iterate through every detail area in the tree's detail area STL list, delete each one and empty that list too.

```
    // Iterate through any stored detail area objects and release
    for ( ; AreaIterator != m_DetailAreas.end(); ++AreaIterator )
    {
        TreeDetailArea * pDetailArea = *AreaIterator;
        if ( pDetailArea ) delete pDetailArea;

    } // Next detail area
    m_DetailAreas.clear();
```

And of course, we do the same for the list of leaves.

```
    // Iterate through the leaves and destroy them
    for ( ; LeafIterator != m_Leaves.end(); ++LeafIterator )
    {
        CBaseLeaf * pLeaf = (CBaseLeaf*)(*LeafIterator);
        if ( pLeaf ) delete pLeaf;

    } // Next Leaf

    m_Leaves.clear();
```

We then release the 3D device that the base tree render system will use. This device was passed in and stored in the constructor. We will see it being used in the following lesson.

```
    // Release other D3D Objects
    if ( m_pD3DDevice ) m_pD3DDevice->Release();

    // Clear required variables
    m_pD3DDevice        = NULL;
}
```

Just remember later when examining the destructors of each derived class, that this base class constructor will always be called too. This is how the main memory for much of the tree gets released. All the derived class has to do in its destructor is delete the root node.

Build - CQuadTree

We finally find ourselves at the point where we can discuss the code that actually builds a tree. The application should call the CQuadTree::Build function only after it has registered all static polygon and detail area data with the tree (using the AddPolygon and AddDetailArea methods inherited from the base class). It is important that at the time this function is called, the object's m_Polygons STL list contains all of the static polygon data we wish to compile into the tree and the m_DetailAreas array contains all of the detail areas that we would like compiled into the tree.

This function is not the recursive function that calls itself until the tree is fully built, but it is the function that allocates the root node and prepares the scene data for the recursive process (invoked by a call to the CQuadTree::BuildTree method). This function has the following tasks to perform:

1. Allocate a new CQuadTreeNode for the root node of the tree.

2. Loop through each polygon in its list and compile an AABB for the data.

3. Loop through each detail area and possibly adjust the AABB computed in step 2 so that all polygon data and detail areas are contained inside the bounding box (this will be the bounding box of the root node).

4. Build a list of all the polygons contained in this node (which will actually be a new temporary list containing all the polygons in the entire scene). Since this is the root node, this list will initially be a copy of the complete m_Polygons list (more on this in a moment). It is this list that will be passed down the tree and divided into four child lists at each node.

5. Compile a list of all the detail area in the root node. This list will initially be an exact copy of the m_DetailAreas list to begin with since this is the root node. As this list is passed down the tree, it will be divided into four lists at each node, one for each child.

6. Call the recursive BuildTree function passing the node pointer (step 1), the list of polygons and detail areas (steps 4 and 5) and the bounding box (steps 2 and 3). The recursive function will then store that passed AABB in the node and will sort the passed polygon and detail area lists into four lists, one for each child it creates. Then it will call itself recursively for each child, passing in the child node and the polygon and detail area lists for that node.

7. When the recursive function returns, call the CBaseTree::PostBuild method. This will instruct the base class to calculate the polygon bounding boxes and initialize the render system. If a valid device pointer was passed to the constructor, the initialization of the render system will be triggered which will also issue a call to the CBaseTree::Repair method to repair all T-junctions that exist in the tree's polygon set.

Before we look at the code, we have one more small matter to address. If a clipped tree is not being compiled, then after the build is finished, the m_Polygons list should and will contain the same polygon pointers that were originally registered with the tree (i.e., the building process will not alter the polygons stored in this list). The CPolygon pointers stored in this list will be passed down the tree and a copy of each polygon pointer will be stored in the leaves in which it is contained. As the polygon data is never clipped or altered, the m_Polygons array will not be altered by the build process and will always contain the polygons contained in the tree even after the build process.

However, if we pass true as the (only) parameter into the Build function, we are informing the tree that we would like the polygon data clipped to the leaf nodes so that no polygon will ever be stored in more than one leaf. We know that when we clip a polygon, we get back two new polygons and the original polygon is deleted and replaced by the two new clips. After the build process has completed in the case of a clipped tree, we do not want the m_Polygons array to contain the same polygons that were registered with the scene as these will no longer describe the polygons being used by the tree. Several polygons in that list will have been deleted the moment they were split, and many new polygons would have been added.

Fortunately there is not much to worry about. As we discussed in the steps above, we copy the polygon pointers into a temporary polygon list for the root node anyway, and it is this list that is passed down the tree. Therefore, in the clipped case, once we have made that temporary copy of the list, we will empty the m_Polygons list so that it contains no data. During the build process, polygons in the temporary list that gets passed to each node will be deleted and split into two new polygons which will all be added to these temporary lists. However, as soon as we find a leaf node, we will be passed a list of clipped polygons that fit exactly inside that leaf. We will then add these pointers to the m_Polygons list as and when they make it into a leaf. That way, as each leaf is created, its new split polygons are added back into the m_Polygons list, leaving us with an m_Polygons list at the end of the build process which contains all the polygons actually being used by the tree. This will be a very different list versus the original polygon set prior to the build process.

Let us now look at the code a section at a time.

The function is passed a single parameter which determines whether a clipped or non-clipped tree should be built. If this boolean is set to false (the default) a non-clipped tree will be built. Take a look at the variables we declare on the stack at the head of the function.

```
bool CQuadTree::Build( bool bAllowSplits /* = false */ )
{
    PolygonList::iterator       PolyIterator = m_Polygons.begin();
    DetailAreaList::iterator    AreaIterator = m_DetailAreas.begin();
    PolygonList                 PolyList;
    DetailAreaList              AreaList;
    unsigned long               i;

        // Reset our tree info values.
    D3DXVECTOR3 vecBoundsMin( FLT_MAX, FLT_MAX, FLT_MAX );
    D3DXVECTOR3 vecBoundsMax( -FLT_MAX, -FLT_MAX, -FLT_MAX );
```

We declare two iterators that will be used to step through the m_Polygons and m_DetailAreas lists. These lists contain the data we wish to compile. However, we also instantiate a local scope polygon list and detail area list that will be used to contain copies of the m_Polygons and m_DetailAreas lists respectively. We then set the vecBoundsMax and vecBoundsMin vectors which will be used to record the size of the root node's volume (an inside out box initially).

Next we allocate the root node of the tree and store its pointer in the CQuadTree member variable. We also copy the value of the bAllowSplits boolean into the m_bAllowSplits member variable.

```
    // Allocate a new root node
    m_pRootNode = new CQuadTreeNode;
    if ( !m_pRootNode ) return false;

    // Store the allow splits value for later retrieval.
    m_bAllowSplits = bAllowSplits;
```

Our next task is to loop through every polygon in the m_Polygons list and adjust the extents of the bounding box such that it is large enough to contain the vertices of all polygons in the scene. At the end of the following loop, vecBoundsMin and vecBoundsMax will represent a box that bounds all the

polygon data in the root node. Note that as we process each CPolygon pointer in m_Polygons and adjust the box to fit it, we add its pointer to the local polygon list declared at the top of the function. Therefore, at the end of this loop, we will have also made a copy of all the polygon pointers contained in m_Polygons into the local PolyList.

```
    // Loop through all of the initial polygons
    for ( ; PolyIterator != m_Polygons.end(); ++PolyIterator )
    {
        // Retrieve the polygon
        CPolygon * pPoly = *PolyIterator;
        if ( !pPoly ) continue;

        // Calculate total scene bounding box.
        for ( i = 0; i < pPoly->m_nVertexCount; ++i )
        {
            // Store info
            CVertex * pVertex = &pPoly->m_pVertex[i];
            if ( pVertex->x < vecBoundsMin.x ) vecBoundsMin.x = pVertex->x;
            if ( pVertex->y < vecBoundsMin.y ) vecBoundsMin.y = pVertex->y;
            if ( pVertex->z < vecBoundsMin.z ) vecBoundsMin.z = pVertex->z;
            if ( pVertex->x > vecBoundsMax.x ) vecBoundsMax.x = pVertex->x;
            if ( pVertex->y > vecBoundsMax.y ) vecBoundsMax.y = pVertex->y;
            if ( pVertex->z > vecBoundsMax.z ) vecBoundsMax.z = pVertex->z;

        } // Next Vertex

        // Store this polygon in the top polygon list
        PolyList.push_back( pPoly );

    } // Next Polygon

    // Clear the initial polygon list if we are going to split the polygons
    // as this will eventually become storage for whatever gets built
    if ( bAllowSplits ) m_Polygons.clear();
```

Notice what we do in the very last line of code show above. If a clipped tree is being built, we empty m_Polygons so that it no longer contains any polygon data. We can do this because we now have a copy of each pointer in the local list. m_Polygons will be repopulated with the clipped polygon data as each leaf node is encountered and created. If a clipped tree is not being built, we do not empty this list since it will be unchanged during the build process.

We now have a bounding box for the root node polygon data but we must also make sure that it is large enough to also bound any registered detail areas. So we now loop through each registered detail area and adjust the root node's bounding box to contain the bounding boxes of all detail areas. As with the above loop, as we process each detail area, we also copy its pointer into the local detail area list (DetailList) so that we have a complete list of detail areas which can be passed into the recursive process.

```
    // Loop through all of the detail areas
    for ( ; AreaIterator != m_DetailAreas.end(); ++AreaIterator )
    {
        // Retrieve the detail area
        TreeDetailArea * pDetailArea = *AreaIterator;
```

```
        if ( !pDetailArea ) continue;

        // Calculate total scene bounding box.
        D3DXVECTOR3 & Min = pDetailArea->BoundsMin;
        D3DXVECTOR3 & Max = pDetailArea->BoundsMax;

        if ( Min.x < vecBoundsMin.x ) vecBoundsMin.x = Min.x;
        if ( Min.y < vecBoundsMin.y ) vecBoundsMin.y = Min.y;
        if ( Min.z < vecBoundsMin.z ) vecBoundsMin.z = Min.z;
        if ( Max.x > vecBoundsMax.x ) vecBoundsMax.x = Max.x;
        if ( Max.y > vecBoundsMax.y ) vecBoundsMax.y = Max.y;
        if ( Max.z > vecBoundsMax.z ) vecBoundsMax.z = Max.z;

        // Store this in the top detail area list
        AreaList.push_back( pDetailArea );

    } // Next Polygon
```

At this point we have a root node pointer, a bounding box large enough to contain all the data contained in that node, and a complete list of all the registered polygons and detail areas stored in the PolyList and AreaList local lists. We are now ready to start the internal build process, so we pass this information into the BuildTree recursive function.

```
    // Build the tree itself
    if ( !BuildTree(m_pRootNode, PolyList, AreaList, vecBoundsMin, vecBoundsMax ))
         return false;
```

This function call will store the passed bounding box in the passed node, which means in its first iteration it will store the bounding box we have just compiled in the root node. It will then divide this bounding box into four quadrants creating the bounding boxes for each of the four child nodes. Then it will classify the polygon list and the detail area list against each bounding box, creating four separate polygon and detail area lists for each child. The four child nodes are then created and the BuildTree function calls itself recursively for each child and the process repeats until the function determines it has been passed a node that should be made into a leaf. This function will be covered in a moment, but for now just know that when it returns program flow back to CQuadTree::Build, the entire hierarchy of nodes will have been created and m_Polygons will contain the polygon data being used by the tree.

At this point, the tree is fully constructed. Before we return program flow back to the application, we issue a call to the CBaseTree::PostBuild function which instructs the base class to calculate the bounding boxes of each polygon and prepare the data for rendering (which includes T-junction repair).

```
    // Allow our base class to finish any remaining processing
    return CBaseTree::PostBuild();
}
```

The recursive CQuadTree::BuildTree function is true core of this system, so let us cover that next.

BuildTree – CQuadTree

BuildTree is the main tree compilation function. It is passed a node, a list of polygons and detail areas that fit in the bounding volume of that node, and the extent vectors of its volume. The function will first assign the passed volume to the passed node. Thus, the first time this function is called, BoundsMin and BoundsMax will be assigned to the root node as its bounding volume. It also means that the first time it is called, PolyList and AreaList will contain the entire scene since the root node's volume bounds everything (assuming you are using the tree for an entire scene of course). In the case of the root node, its bounding volume was calculated in the previously discussed function and passed in along with the node. For all other nodes however, this function will generate the bounding boxes and polygon lists for each of its child nodes and call itself recursively. Therefore, the second time this function is called it will be called by the previous instance of the function. The node pointer passed will not be the root, it will be the first child of the root. The PolyList and AreaList will contain only the geometry that was contained in the root node's first child, and BoundsMax and BoundsMin will be the bounding volume of that child node quadrant.

Let us first look at the variables that are allocated on the stack. We will need to be able to iterate through the passed polygon and detail area lists so that we can divide them into four sub-lists (one per child), so we instantiate two iterators for this purpose. We will also need four temporary lists to hold the polygon data for each child node and the detail areas for each child node, so you can see that we instantiate an array of four PolygonLists (an STL list of CPolygons) and four DetailAreaLists (an STL list of detail area structures). We allocate two D3DXPLANE structures that will be used to create and store the two split planes at this node that divide the node into four quadrants. We will see these being used in a moment.

Again, the first thing we do in the following code is assign the passed volume to the passed node.

```
bool CQuadTree::BuildTree( CQuadTreeNode * pNode,
                           PolygonList PolyList,
                           DetailAreaList AreaList,
                           const D3DXVECTOR3 & BoundsMin,
                           const D3DXVECTOR3 & BoundsMax )
{
    D3DXVECTOR3                 Normal;
    PolygonList::iterator       PolyIterator;
    DetailAreaList::iterator    AreaIterator;
    CPolygon                  * CurrentPoly, * FrontSplit, * BackSplit;
    PolygonList                 ChildList[4];
    DetailAreaList              ChildAreaList[4];
    CCollision::CLASSIFYTYPE    Location[2];
    D3DXPLANE                   Planes[2];
    unsigned long               i;
    bool                        bStopCode;

    // Store the bounding box properties in the node
    pNode->BoundsMin = BoundsMin;
    pNode->BoundsMax = BoundsMax;
```

Before we start subdividing space we need to determine whether or not the leaf is so small or its polygon list contains so few polygons that this node should be made a terminal node (a leaf). We discussed earlier that there are three stop codes that if true, prevent us creating any more children. We simply store the passed polygon data in the node, making it a leaf. Therefore, we first calculate the diagonal length of the current node's bounding box and test this against the m_fMinLeafSize stop code.

```
    // Calculate 'Stop' code
    D3DXVECTOR3 vecLeafSize = D3DXVECTOR3( BoundsMax.x - BoundsMin.x,
                                           0,
                                           BoundsMax.z - BoundsMin.z );

    bStopCode  = (AreaList.size() == 0 && PolyList.size() == 0) ||
                 (AreaList.size() <= m_nMinAreaCount  &&
                  PolyList.size()<= m_nMinPolyCount ) ||
                  D3DXVec3Length( &vecLeafSize ) <= m_fMinLeafSize;
```

Let us analyze the multiple conditionals being used above. First, if there are no detail areas in the passed area list and no polygons in the passed polygon list, the node is totally empty and the boolean bStopCode will be set to true. In that case, we will create a leaf here. This boolean is also set to true if the number of detail areas is less then or equal to the m_nMinAreaCount stop code or if the number of polygons is less than or equal to the minimum polygon count amount. So, if we have too few polygons and too few detail areas in this node (as defined by the stop codes), we set bStopCode to true since we wish to halt subdivision and make this node a leaf. The final test that sets bStopCode to true happens when the length of the diagonal vector from the node's extents is less than or equal to the minimum leaf size stop code. Notice that because we are using a quad-tree, all nodes will have the same height and we do not factor in the y components of the extents (we set the vecLeafSize vector's Y component to zero). For a quad-tree, the diagonal vector projected on to the XZ plane is all we are concerned with.

At this point we have a boolean that tells us whether we should continue to divide this node into four children or whether we should just make a new leaf here and return.

In the next section of code we see what happens if this node is to become a leaf. We first allocate a new CBaseLeaf structure, passing in a pointer to the current tree instance, and then loop through every polygon in the passed list and add it to the leaf's polygon array using the CBaseLeaf::AddPolygon function. At the end of this loop all the polygons that made it into the node will be stored in a new CBaseLeaf.

```
    // If we reached our stop limit, build a leaf here
    if ( bStopCode )
    {
        // Build a leaf
        CBaseLeaf * pLeaf = new CBaseLeaf( this );
        if ( !pLeaf ) return false;

        // Store the polygons
        for ( PolyIterator = PolyList.begin();
              PolyIterator != PolyList.end();
              ++PolyIterator )
        {
            // Retrieve poly
```

```
            CurrentPoly = *PolyIterator;
            if ( !CurrentPoly ) continue;

            // Add to full tree polygon list ONLY if splitting is allowed
            if ( m_bAllowSplits ) AddPolygon( CurrentPoly );

            // Also add a reference to the leaf's list
            pLeaf->AddPolygon( CurrentPoly );

        } // Next Polygon
```

Notice something very important in the above code. If a clipped tree is being built, we need to repopulate the tree's m_Polygons array which was emptied at the start of the build process. We do this using the CBaseTree::AddPolygon function. This way, when all leaves have been created for the tree, the m_Polygons list will be fully populated with all the polygon fragments that made it into each clipped leaf. Obviously, we do not add the polygons to the tree's list if we are not building a clipped tree as they are already stored there.

Our next task is to loop through each detail area that made it into this node's list and add each of those to the new leaf's detail area array. The CBaseLeaf::AddDetailArea method which we covered earlier will be used to perform this task.

```
        // Store the area lists
        for ( AreaIterator = AreaList.begin();
              AreaIterator != AreaList.end(); ++AreaIterator )
        {
            // Retrieve detail area item
            TreeDetailArea * pDetailArea = *AreaIterator;
            if ( !pDetailArea ) continue;

            // Add a reference to the leaf's list
            pLeaf->AddDetailArea( pDetailArea );

        } // Next Polygon
```

At this point we have a new leaf and it contains (in its internal arrays) a list of all the CPolygon pointers and detail area pointers that made it into this node. We will now assign the node's Leaf pointer to point at our new leaf object and set the bounding box of the leaf to be the same as the node's bounding box (using the CBaseLeaf::SetBoundingBox method to perform this task). Finally, with the leaf attached the node and its internal structures populated with the node's data, we add this leaf object's pointer to the tree's leaf list for easier access and cleanup. We implemented the CBaseTree::AddLeaf method earlier to take care of adding a leaf pointer to the tree's leaf list.

At this point the leaf has been created and the node's child pointers will still be NULL, and that is how they should remain. This is an end of branch of the tree and we can simply return because our terminal node has been created. This halts the recursive process down this particular branch of the tree.

```
        // Store pointer to leaf in the node and the bounds
        pNode->Leaf = pLeaf;

        // Store the bounding box in the leaf
```

```
        pLeaf->SetBoundingBox( pNode->BoundsMin, pNode->BoundsMax );

        // Store the leaf in the leaf list
        AddLeaf( pLeaf );

        // We have reached a leaf
        return true;

    } // End if reached stop code
```

If we reach this point in the code, then it means we are not in a leaf node and as such, we need to divide the node's volume into four quadrants so that we can build appropriate polygon and detail area lists as well as new bounding volumes.

These four quadrants describe the bounding volumes of the four child nodes we will create and attach to this node.

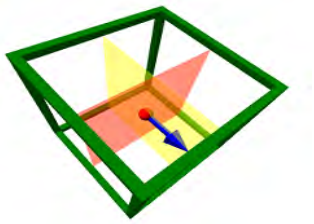


Figure 14.78

As discussed earlier in the lesson, we will determine which quadrant each polygon is in by creating two split planes that are aligned with the world X and Z axes and pass through the center point of the current node's bounding volume. The two split planes are shown in Figure 14.78. We will classify each polygon in this node's list against these two planes to determine in which quadrants they exist and to which child polygon list they should be assigned.

The next section of code generates the two split planes and stores them in the two element local Planes array. Plane[0] has its normal pointing in the direction of the Z axis (XY plane) and Plane[1] is aligned with the world YZ plane with its normal pointing down the X axis. The point on each plane which helps fully describe the plane to the D3DXPlaneFromPointNormal function is the same for each plane: the center point of the node's volume calculated by adding its maximum extents to its minimum extents and dividing the result by two. The D3DXPlaneFromPointNormal function will return a D3DXPLANE structure that describes the plane, not in point/normal format, but in normal/distance format.

```
    // Generate the two split planes
    D3DXPlaneFromPointNormal( &Planes[0], &((BoundsMin + BoundsMax) / 2.0f),
                                          &D3DXVECTOR3( 0.0f, 0.0f, 1.0f ) );

    D3DXPlaneFromPointNormal( &Planes[1], &((BoundsMin + BoundsMax) / 2.0f),
                                          &D3DXVECTOR3( 1.0f, 0.0f, 0.0f ) );
```

> **Note:** Because we have to give some descriptive names to the quadrants of our node to make things a little easier to explain, we will refer to each quadrant using a name that is relative to an imaginary individual positioned at the center of the node looking in the direction of Plane[0]'s normal. As Plane[0] faces down the positive Z axis, the quadrants behind this plane will be referred to as 'BehindLeft' and 'BehindRight'. The two quadrants in front of Plane[0] will be labelled 'InfrontLeft' and 'InfrontRight'.

If we were only supporting non-clipped trees, then our next task would be to loop through each polygon in the node's list, classify it against both planes and use the two results to determine which quadrants the polygon falls into. The polygon would then have its pointer added to the appropriate lists for those quadrants (remember that we allocated an array of four empty polygon lists on the stack which we will

fill in this function with the polygon contained in each quadrant). However, if the m_bAllowSplits member is set to true, it means the application would like this tree built so that no polygon is ever spanning node boundaries. As such, all the polygons in the node's list that are spanning any of the two node planes should be split. The original polygon that was spanning the plane should then be deleted from the list and replaced with the two new polygons that it was split into.

The next section of code is executed only in the case when a clipped tree is being built. It loops through each polygon in this node's list and classifies each polygon against the current plane being tested. If a polygon is found to be spanning the current plane we are testing, we split it into two, delete it, set its pointer entry in the list to NULL and then add the two new child polygons to the list. These child polygons will possibly be split again when the second plane is tested. We discussed code almost identical to this at the beginning of this lesson, so you should have no problem understanding how it works. Notice that the task is made very easy due the functions we have previously written (CPolygon::Split and CCollision::PolyClassifyPlane).

```
    // Split all polygons against both planes if required
    if ( m_bAllowSplits )
    {
        for ( i = 0; i < 2; ++i )
        {
            for ( PolyIterator = PolyList.begin();
                  PolyIterator != PolyList.end(); ++PolyIterator )
            {
                // Store current poly
                CurrentPoly = *PolyIterator;
                if ( !CurrentPoly ) continue;

                // Classify the poly against the first plane
                Location[0] = CCollision::PolyClassifyPlane
                            (
                               CurrentPoly->m_pVertex,
                               CurrentPoly->m_nVertexCount,
                               sizeof(CVertex),
                               (D3DXVECTOR3&)Planes[i],
                               Planes[i].d
                            );


                if ( Location[0] == CCollision::CLASSIFY_SPANNING )
                {
                    // Split the current poly against the plane,
                    // delete it and set it to NULL in the list
                    CurrentPoly->Split( Planes[i], &FrontSplit, &BackSplit );
                    delete CurrentPoly;
                    *PolyIterator = NULL;

                    // Add these to the end of the current poly list
                    PolyList.push_back( FrontSplit );
                    PolyList.push_back( BackSplit );

                } // End if Spanning

            } // Next Polygon
```

```
        } // Next Plane

    } // End if allow splits
```

At this point, if we are building a clipped tree, any polygons in the list that were spanning the node planes will have been deleted and replaced with polygons that fit neatly into each leaf. If this is not a clipped tree, then it does not matter if a polygon is spanning a node plane as we will just assign it to all the child node lists in which it belongs.

Our next task is to loop through each polygon in the list and classify it against both of the node's split planes. We store the classification results in a local ClassifyType array called Location.

```
    // Classify the polygons and sort them into the four child lists.
    for ( PolyIterator = PolyList.begin();
          PolyIterator != PolyList.end(); ++PolyIterator )
    {
        // Store current poly
        CurrentPoly = *PolyIterator;
        if ( !CurrentPoly ) continue;

        // Classify the poly against the planes
        Location[0] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                     CurrentPoly->m_nVertexCount,
                                                     sizeof(CVertex),
                                                     (D3DXVECTOR3&)Planes[0],
                                                     Planes[0].d );

        Location[1] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                     CurrentPoly->m_nVertexCount,
                                                     sizeof(CVertex),
                                                    (D3DXVECTOR3&)Planes[1],
                                                     Planes[1].d );
```

At this point we know the current polygon's relationship to both the split planes and have the results stored, so we can figure out which quadrant it is in and which of the child polygon lists to add it to. Location[0] tells us whether it is in the back or front halfspace of the node, while Location[1] tells us in which halfspace within that halfspace (left or right) the polygon belongs.

Since the first plane split plane is aligned to the world XY plane, we know that if it is spanning the plane (only possible in the non-clipped tree case) or it is behind this plane, then it will need to have its pointer assigned to either the BehindLeft or BehindRight lists. Once we know it is behind the XY plane, we then test its classification against the YZ plane. If it is behind this plane then it must be in the BehindLeft quadrant; otherwise it must be in the BehindRight quadrant. In either case, we add it to the relevant list. ChildList[0] will contain the polygons found to be in the BehindLeft quadrant and ChildList[1] is assigned polygons that are contained in the BehindRight quadrant.

```
        // Position relative to XY plane
        if ( Location[0] == CCollision::CLASSIFY_BEHIND ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
```

```
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildList[0].push_back( CurrentPoly );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_ONPLANE ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildList[1].push_back( CurrentPoly );

        } // End if behind or spanning
```

Notice in the above code that if a polygon is found to lie on the plane YZ plane we assign it to the node in front of the plane. An on plane polygon is essentially on the border between two nodes and we need to assign it somewhere. It does not really matter which node we choose to assign it to because if that polygon is visible, it would mean that the leaves both in front and behind that node would always be collected anyway. Likewise, if a collision query is performed and the swept sphere did intersect that polygon, it would automatically mean that it is spanning those leaves and those leaves would be collected.

If the above code was not executed it must mean the polygon is in front of the XY node plane (Plane[0]), so we must include similar code that will add the polygon to either front left or right quadrants. However, the next section of code is also executed in the spanning case as well, just as the above code was. This means that in the case of a non-clipped tree where we have a polygon spanning the XY plane, both code blocks will be executed and the polygon will be added to nodes both in front and behind this plane. This is correct, because if a polygon spans the boundaries of two nodes (which is only possible at this point in the non-clipped tree), we wish to add it to both nodes.

The next section of code handles the list assignments for polygons that are either spanning or are contained in the front halfspace of the XY plane. It is executed if the polygon is in front of the XZ plane or (as discussed above) if the polygon is on the plane. For our spatial trees, we can safely treat the on plane case as being identical to the in front case.

```
        if ( Location[0] == CCollision::CLASSIFY_INFRONT ||
             Location[1] == CCollision::CLASSIFY_ONPLANE ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildList[2].push_back( CurrentPoly );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_ONPLANE ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                 ChildList[3].push_back( CurrentPoly );

        } // End if in-front or on-plane

    } // Next Triangle
```

We can see above that ChildList[2] is the list compiled for the quadrant that is in front of the XY plane and behind the YZ plane (the back left quadrant) and ChildList[3] is for polygons that are in front of both planes (the back right quadrant).

At this point we have four polygon lists (one for each quadrant) and we now need to do the same thing with the node detail areas. Notice that the following code is identical to that we saw above with the exception that we are using our new CCollision::AABBClassifyPlane method to calculate the classification between the AABB and each plane.

```
    // Classify the areas and sort them into the child lists.
    for ( AreaIterator = AreaList.begin();
          AreaIterator != AreaList.end(); ++AreaIterator )
    {
        // Store current area
        TreeDetailArea * pDetailArea = *AreaIterator;
        if ( !pDetailArea ) continue;

        // Classify the area against the planes
        Location[0] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)Planes[0],
                                                     Planes[0].d );

        Location[1] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)Planes[1],
                                                     Planes[1].d );
```

Once again, the same logic is used to determine in which quadrant the AABB is contained and to which list it should be added.

```
        // Position relative to XY plane
        if ( Location[0] == CCollision::CLASSIFY_BEHIND ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                            ChildAreaList[0].push_back( pDetailArea );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                            ChildAreaList[1].push_back( pDetailArea );

        } // End if behind or spanning
        if ( Location[0] == CCollision::CLASSIFY_INFRONT ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                            ChildAreaList[2].push_back( pDetailArea );
```

```
            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildAreaList[3].push_back( pDetailArea );

        } // End if in-front or on-plane

    } // Next Detail Area
```

We are very nearly done. We have four lists of polygons that describe the polygons that fit in each quadrant of the node, and we have four lists of detail objects that also are contained (or partially contained) in each quadrant. We know at this point that we want each quadrant in this node to be represented by four new child nodes. As the BuildTree function will be called for each of these nodes and expects to be passed the node's bounding volume, we will have to create the AABB for each child node in this function.

In the next and final section of code, we set up a loop that will iterate for each child. Depending on which iteration of the loop we are on we will calculate the bounding box for that node and store it in temporary vectors NewBoundsMin and NewBoundsMax. The box computation is simple since the mid-point of the parent node will describe the location where all child volumes meet in the center of the node. We can see for example that in the first iteration of the loop we are building the volume for the BehindLeft child. Its minimum extents vector is simply the minimum extents vector of the parent node and its maximum extents vector is in the center of the parent node along the x and z axes. All quad-tree nodes inherit their height from the root node, so we can see that all nodes will always have a maximum y extents component of BoundsMax.y and will always have a minimum y extents component of BoundsMin.y. Once we have the bounding box for the current node we are processing, we allocate a new CQuadTreeNode structure and store its pointer in the parent node's child pointer list. The function then calls itself recursively passing in the new child node, the bounding box of that node that we have just calculated, and the polygon and detail area lists we compiled for its quadrant.

```
    // Build each of the children here
    for( i = 0; i < 4; ++i )
    {
        // Calculate child bounding box values
        D3DXVECTOR3 NewBoundsMin,
                    NewBoundsMax,
                    MidPoint = (BoundsMin + BoundsMax) / 2.0f;
        switch( i )
        {
            case 0: // Behind Left
                NewBoundsMin = BoundsMin;
                NewBoundsMax = D3DXVECTOR3( MidPoint.x, BoundsMax.y, MidPoint.z);
                break;

            case 1: // Behind Right
                NewBoundsMin = D3DXVECTOR3( MidPoint.x, BoundsMin.y, BoundsMin.z );
                NewBoundsMax = D3DXVECTOR3( BoundsMax.x, BoundsMax.y, MidPoint.z);
                break;

            case 2: // Infront Left
                NewBoundsMin = D3DXVECTOR3( BoundsMin.x, BoundsMin.y, MidPoint.z );
                NewBoundsMax = D3DXVECTOR3( MidPoint.x, BoundsMax.y, BoundsMax.z );
                break;
```

```
            case 3: // Infront Right
                NewBoundsMin = D3DXVECTOR3( MidPoint.x, BoundsMin.y, MidPoint.z );
                NewBoundsMax = BoundsMax;
                break;

        } // End Child Type Switch

        // Allocate child node
        pNode->Children[i] = new CQuadTreeNode;
        if ( !pNode->Children[i] ) return false;

        // Recurs into this new node
        BuildTree( pNode->Children[i],
                   ChildList[i],
                   ChildAreaList[i],
                   NewBoundsMin,
                   NewBoundsMax );

        // Clean up
        ChildList[i].clear();
        ChildAreaList[i].clear();

    } // Next Child

    // Success!
    return true;
}
```

In each iteration of the node generation loop above, the BuildTree function is passed a child node. As this function will perform all the same steps on the child node, and so on down the tree, until a leaf is located, we can imagine that once this function returned for the first child node, the entire branch of the tree that starts at that node will have been created. We then process the remaining three nodes in the loop and our job is done. Notice at the bottom of the node loop that once the BuildTree method returns for a given child, the polygon list and detail area list we compiled for it earlier in the function are no longer needed, so we release them. When this loop exits, every node and leaf under the current node being processed will have been built, so we can return. If we are in the instance of the BuildTree method that was called for the root node for example, the tree will have been completely built and program flow will pass back to the Build method which we saw previously.

You have just written a quad-tree compiler and implemented all the steps involved in the build process. The really nice thing about this is that the build functions for our oct-tree and kD-tree are almost identical. The only difference is that they split the node with a different number of planes and create a different number of child nodes. This means that we will be able to examine the build methods of the other tree with very little explanation.

Although we have covered all the code involved in the build process, there are still several query methods we must implement in the derived class to allow the application (or any user) to collect leaves based on intersections. We will examine these queries next before moving on to the other tree types.

14.17.3 The Quad-Tree Query Methods

The query functions for any spatial tree are what make the tree useful. The CollectLeavesAABB method for example is used by the application to link dynamic objects to leaves. The application can simply pass the bounding volume of the dynamic object into the CollectLeavesAABB method and have a list filled with pointers to all the leaves the volume intersected. This is obviously very handy for tasks like visibility processing since the dynamic object only has to be rendered if the IsVisible method of one of these leaves returns true.

The CollectLeavesAABB method is not just used by the application. Recall that CBaseTree demands that it is implemented in the derived class because it is called to speed up the T-junction repair process. For each polygon currently being repaired, it was used to return a list of leaves whose volumes intersect the AABB of the polygon. Only the polygons contained in these leaves would need to be tested against each other at the edge/vertex level.

For each of our tree classes we will implement two query functions which allow us to collect leaves from the tree using different primitives. The CollectLeavesAABB method sends an AABB down the tree and adds the pointer of any leaf it intersects to the passed list. The CollectLeavesRay method passes a ray down the tree and adds the pointers to any leaves the ray intersects to the passed leaf list. Fortunately, both functions are quite small due to the fact that they use the collision routines we added to CCollision earlier in this lesson. You should feel free to add query routines of your own for different primitive types (spheres would be another good choice) and extend the ISpatialTree interface as needed.

CollectLeavesAABB – CQuadTree

The CollectLeavesAABB method is a wrapper around the first call to the CollectAABBRecurse method (the recursive method that steps through the tree and adds any leaves it encounters to the passed leaf list). The CollectLeavesAABB method just starts the recursive process at the root node of the quad-tree. Here is the code.

```
bool CQuadTree::CollectLeavesAABB( LeafList & List,
                                   const D3DXVECTOR3 & Min,
                                   const D3DXVECTOR3 & Max )
{
    // Trigger Recursion
    return CollectAABBRecurse( m_pRootNode, List, Min, Max );
}
```

The method is passed three parameters. The first should be of type LeafList, which you will recall is a type definition for an STL linked list that contains ILeaf pointers. The list is passed by reference and is assumed to be empty, since it will be the job of the CollectAABBRecurse method to fill it with leaves. The second and third parameters are the extents of the AABB we would like to query the tree with (e.g., the AABB of a dynamic object).

When the CollectAABBRecurse method returns, the leaf list will contain all the pointers to leaves that were intersected by the passed AABB. Of course, the core of the query process is found inside the CollectAABBRecurse method, which we will examine next.

CollectAABBRecurse - CQuadTree

This function is passed a pointer to the node it is currently visiting, which will be a pointer to the root node the first time it is called by CollectLeavesAABB. It is also passed a leaf list which it should fill with any leaves found to be intersecting the query volume described by parameters Min and Max. The function also accepts a final boolean parameter (set to false by default) called bAutoCollect. Because this parameter is optional, and is not passed by the previous function, this will always be set to zero when it is first called for the root node. This boolean will be tracked as the tree is traversed and will be set to true once we find a node that is completely contained in the query volume.

You will recall from earlier in the lesson that we wrote a CCollision::AABBIntersectAABB method which took a boolean reference as its first parameter. The function returns true if the two boxes intersect but the boolean passed in the first parameter will also be set to true if the second box is fully contained in the first. We use this to optimize our leaf collection process because, if we find at any node in the tree that has its box fully contained inside the query volume, there is no point in performing the same AABBIntersectAABB test as we visit each of its children. Since all the children will be contained in the parent node's box, which is itself fully contained in the query volume, we know that all nodes and leaves under that node must be contained in the query volume as well. Therefore, as soon as the boolean is set to true, any further AABB tests on the children of that node will be abandoned; we will simply traverse down that branch of the tree and add leaves to the list as we find them.

The function uses the CCollision::AABBIntersectAABB method we wrote earlier in this lesson to test if the passed AABB intersects the AABB of the node currently being visited. If it does not, then this node and all of its child cannot possibly intersect the query volume, so we can return false immediately and stop traversing down this branch of the tree. Again, we only do the AABB intersection test if the bAutoCollect parameter is not set to true. If it is, then we can just proceed to collect leaves without further testing.

```
bool CQuadTree::CollectAABBRecurse( CQuadTreeNode * pNode,
                                    LeafList & List,
                                    const D3DXVECTOR3 & Min,
                                    const D3DXVECTOR3 & Max,
                                    bool  bAutoCollect /*=false*/ )
{
    bool  bResult = false;
    ULONG i;

    // Validate parameters
    if ( !pNode ) return false;

    // Does the specified box intersect this node?
    if ( !bAutoCollect && !CCollision::AABBIntersectAABB( bAutoCollect,
                                                          Min,
```

```
                                                        Max,
                                                        pNode->BoundsMin,
                                                        pNode->BoundsMax,
                                                        false,
                                                        true,
                                                        false ) ) return false;
```

Notice that we pass our boolean variable as the first parameter to the AABB intersection method so that the function can set its value to true in the event of box 2 being completely contained in box 1. As this boolean will be passed into the child node traversals this will allow us to avoid needlessly performing AABB tests at those child nodes. The extents vectors of the query volume are passed as the next two parameters followed by the extents of the node's bounding box in the second pair of parameters. Remember that the three boolean parameters at the end of this method's parameter list indicate whether we would like to ignore any axes during the test. For a quad-tree, where every node has the same height, we pass true as the bIgnoreY parameter so that the AABB intersection test is performed only on the XZ plane. It assumes that the quad-tree nodes are infinitely tall since there is no vertical partitioning in a vanilla quad-tree.

If the above collision code did not force an early return from the function, it means the query volume does intersect the node's volume. If the node's Leaf member is not set to NULL then this node is a leaf node and we add its leaf pointer to the passed list. As a leaf is a terminal node, it has no children for us to traverse and as such, our job is done and we can return.

```
    // Is there a leaf here, add it to the list
    if ( pNode->Leaf ) { List.push_back( pNode->Leaf ); return true; }
```

If the above condition was not true then it means the node intersects the query volume but the node is not a leaf node. In this case, we want to traverse into each of its four children. The function recursively calls itself for each child, making sure that it passes the original list and query volume extents down to the children. Notice in the following code how we pass the bAutoCollect boolean value into the child nodes so that if it is set to true, the AABB tests will not be performed on the child nodes (they are assumed to be inside the query volume). The remainder of the function is shown below.

```
    // Traverse down to children
    for ( i = 0; i < 4; ++i )
    {
        if ( CollectAABBRecurse( pNode->Children[i], List, Min, Max, bAutoCollect))
        bResult = true;

    } // Next Child

    // Return the 'was anything added' result.
    return bResult;
}
```

It is hard to believe that such a small function could be so powerful and useful. But thanks to spatial partitioning this is indeed the case. The function to query a ray against the tree is equally as simple, as we will see next.

CollectLeavesRay – CQuadTree

The CollectLeavesRay method can be called by the application to fill a list of all the leaves intersected by the passed ray. The method is a wrapper around the call to the CollectRayRecurse method for the root node. It is the CollectRayRecurse method which performs the traversal and collision logic.

```
bool CQuadTree::CollectLeavesRay( LeafList & List,
                                  const D3DXVECTOR3 & RayOrigin,
                                  const D3DXVECTOR3 & Velocity )
{
    return CollectRayRecurse( m_pRootNode, List, RayOrigin, Velocity );
}
```

The method is passed three parameters, the leaf list that we would like to have filled with leaves which intersect the ray, the ray origin, and the ray delta vector.

CollectRayRecurse – CQuadTree

This method is almost identical to the CollectLeavesRecurse method discussed above. It recursively calls itself visiting each node. At the current node being visited it performs a ray/box intersection test between the passed ray and the node's AABB. We use our new CCollision::RayIntersectAABB method for this and once again pass in a boolean parameter to indicate that we would like to ignore the y dimensions in the test and assume the box to be infinitely high. Although the RayIntersectAABB method returns the *t* value of intersection in its fourth parameter, we are only interested in a true or false result in this case and as such, the *t* value is ignored.

```
bool CQuadTree::CollectRayRecurse(  CQuadTreeNode * pNode,
                                    LeafList & List,
                                    const D3DXVECTOR3 & RayOrigin,
                                    const D3DXVECTOR3 & Velocity )
{
    bool  bResult = false;
    ULONG i;
    float t;

    // Validate parameters
    if ( !pNode ) return false;

    // Does the ray intersect this node?
    if ( !CCollision::RayIntersectAABB( RayOrigin, Velocity,
                                        pNode->BoundsMin,
                                        pNode->BoundsMax,
                                        t,
                                        false,
                                        true,
                                        false ) ) return false;

    // Is there a leaf here, add it to the list
```

```
    if ( pNode->Leaf ) { List.push_back( pNode->Leaf ); return true; }

    // Traverse down to children
    for ( i = 0; i < 4; ++i )
    {
        if ( CollectRayRecurse( pNode->Children[i],
                                List,
                                RayOrigin,
                                Velocity ) ) bResult = true;

    } // Next Child

    // Return the 'was anything added' result.
    return bResult;
}
```

As you can see, if the ray does not intersect the box, then it also means it cannot possibly intersect any children of the node, so we can return false are reject the rest of this branch of the tree. If the ray does intersect the box and the node is a leaf, we add its pointer to the leaf list and return. Finally, if the node is intersected by the ray but it is not a leaf node, we traverse into each of the node's children.

14.17.4 The Quad-Tree DebugDraw Routines

As discussed earlier, we have implemented some deliberately simple debug drawing routines that allow an application to request that the bounding boxes of the leaf nodes be rendered. These routines are far from optimal and are not designed to ship in commercial code. They are written help you solve any problems or choose a nice leaf size for your tree. These functions are small since we included the code that renders a box in a CBaseTree method so that all derived classes can easily implement their debug routines. We also implemented a screen tint function in CBaseTree which is called to tint the screen red when the camera enters a leaf that contains polygon or detail area data.

DebugDraw - CQuadTree

The only debug routine the application has to call is the DebugDraw routine. It should be called only after the entire scene has been rendered. It is passed the camera the application is currently using, which it passes along to the recursive function, DebugDrawRecurse. It is this function which walks the tree and renders the bounding boxes. The code to the CQuadTree::DebugDraw method is shown below.

```
void CQuadTree::DebugDraw( CCamera & Camera )
{
    // Simply start the recursive process
    DebugDrawRecurse( m_pRootNode, Camera, false );

    if ( DebugDrawRecurse( m_pRootNode, Camera, true ) )
         DrawScreenTint( 0x33FF0000 );
}
```

Notice that the DebugDrawRecurse method is called twice since we wish to render the boxes in two passes through the hierarchy. In the first pass, we want to render all the leaf nodes the camera is not currently contained in, which is why we pass false as the final parameter. These leaf nodes will be rendered blue, with Z testing enabled. This means the rendered boxes will be obscured by any geometry that is closer to the camera, just like normal scene rendering. The second time the method is called we pass true as this final parameter, which means we are only interested in rendering the leaf the camera is currently contained in. For these leaves, Z testing will be disabled so that the lines of the box overwrite anything already in the frame buffer, even if that geometry is closer to the camera. This allows us to always see all the lines of the bounding box of the leaf in which you are currently contained. If the current camera leaf has no geometry or detail areas assigned to it (it is an empty leaf) the box is rendered in green and the function return false. If it does have geometry or detail area data, the box lines are rendered in red and the function returns true. We can see in the above code that when this is the case, we also call the CBaseTree::DrawScreenTint function (covered earlier in the lesson) to tint the screen red (alpha value of 0x33).

DebugDrawRecurse - CQuadTree

This method visits each node in the tree and determines whether its box should be rendered and if so, in what color. If the boolean parameter is set to false, the function will render a leaf box when the camera position is not contained in it. If the boolean parameter is set to true, the method will traverse the tree searching only for the camera leaf node and render that box in a different manner than the other boxes. The method will also return true if the boolean parameter is set to true and the camera is contained in a leaf that has either polygon data or detail area assigned to it.

As this method is only interested in rendering leaf nodes, it first tests to see if the node it is currently visiting is a leaf node. If it is, then it uses our new CCollision::PointInAABB test to see if the camera position is contained inside the AABB of the leaf. We once again pass true for the bIgnoreY parameter in the quad-tree case so that the test is essentially performed using only the x and z components of the box and the point. We store the result of this test in the bInLeaf local variable. It will be set true only if the node this function is currently visiting is a leaf and the camera position is within its AABB.

```
bool CQuadTree::DebugDrawRecurse( CQuadTreeNode * pNode,
                                  CCamera & Camera,
                                  bool bRenderInLeaf )
{
    ULONG i;
    bool  bDrawTint = false;

    // Render the leaf
    if ( pNode->Leaf )
    {
        ILeaf * pLeaf = pNode->Leaf;
        bool bInLeaf = CCollision::PointInAABB( Camera.GetPosition(),
                                                pNode->BoundsMin,
                                                pNode->BoundsMax,
                                                false, true, false );
```

At this point, if bInLeaf is set to true and the bRenderInLeaf parameter was also set to true, it means this method is in the mode that renders only the leaf the camera is in. As bInLeaf is true, we have found that leaf so we must render it. We first fetch the polygon count and the detail area count from the leaf. If both are set to zero then it means the camera is in an empty leaf and we set its color to green. Otherwise, we set the color of the box lines to red. If the leaf is not empty, we also set the bDrawTint value to true. This is the value that will be returned from the function. It is set to false at the start of the function by default, so this function will only ever return true if the camera is in a non-empty leaf node and if the function is in the mode that renders only the camera leaf. We then call the CBaseTree::DrawBoundingBox method to render the lines of the leaf's box.

```
        if ( bRenderInLeaf && bInLeaf )
        {
            ULONG Color = 0xFFFF0000; // Red by default

            // Is there really anything in this?
            if ( pLeaf->GetPolygonCount() == 0
                 && pLeaf->GetDetailAreaCount() == 0 )
                Color = 0xFF00FF00;
            else
                bDrawTint = true;

            // Draw the bounding box
            DrawBoundingBox( pNode->BoundsMin, pNode->BoundsMax, Color, false );

        } // End if we should draw a red (inside) box here
```

Notice in the above call to DrawBoundingBox that we pass in the extents of the node's bounding box and the color we calculated based on the leaf's empty status. We also pass in false as the final parameter which instructs the method to render this box without Z testing enabled. This means all the visible lines of the box (those in the frustum) will always be rendered over the top of anything contained in the frame buffer. This will make it much easier to see the extents of the box the camera is currently in without worrying about the lines of the box being obscured by nearby geometry.

We have handled the case for when the function is in 'draw camera leaf only' mode. The next section of code is executed for the 'draw everything except the camera leaf' mode and the current node is not a node the camera is contained in. It uses the CBaseTree::DrawBoundingBox method to draw a blue box, only this time we pass true as the final parameter so that the depth buffer is enabled and the box is rendered in the normal way.

```
        // Draw blue box?
        if ( !bRenderInLeaf && !bInLeaf )
        {
            DrawBoundingBox( pNode->BoundsMin, pNode->BoundsMax, 0xFF0000FF,true );

        } // End if we should draw a blue box (outside) here
```

Finally, as detail areas are not something we can usually see, we will also render a box around any detail areas that might be contained in the current leaf node. As you can see, we simply set up a loop to extract each detail area assigned to the current leaf and render a blue box using the detail area's bounding box. For detail areas, we render with depth testing disabled so that they are not obscured by nearby geometry.

```
        for ( i = 0; i < pLeaf->GetDetailAreaCount(); ++i )
        {
            TreeDetailArea * pDetailArea = pLeaf->GetDetailArea( i );
            if ( !pDetailArea ) continue;

            // Now let's render the detail area
            DrawBoundingBox( pDetailArea->BoundsMin,
                             pDetailArea->BoundsMax,
                             0xFF00FF00,
                             false );

        } // Next Detail Area

    } // End if not a leaf
```

We have now seen all the code that is executed if the current node is a leaf node. If it is not, then this is a non-terminal node that will have four children, so we better visit those too.

```
    // Step down to children.
    for ( i = 0; i < 4; ++i )
    {
        if ( pNode->Children[i] )
        {
            if ( DebugDrawRecurse( pNode->Children[i], Camera, bRenderInLeaf ) )
                 bDrawTint = true;

        } // End if child exists

    } // Next child

    // Return whether we should draw the tint
    return bDrawTint;

}
```

If this function returns true, it means the camera is in a leaf that contains geometry or detail areas and the parent function (DebugDraw) tints the screen slightly red when this is the case.

14.17.5 Quad-Tree Conclusion

We have now covered all the code to build and query the quad-tree for collision and spatial queries. In the next lesson we will add code to CBaseTree (and one or two small functions to the derived classes) that will exist to implement an efficient hardware rendering system. The focus of this lesson however has been on using the tree for spatial queries and we have now successfully created a quad-tree. We will now move on to discuss the implementation of the other tree types, which should be quite easy now that we have covered the quad-tree code and the concepts are all very similar.

14.18 The Y-Variant Quad-Tree

An alternative approach to working with quad-trees is to factor in the Y component at each node such that the bounding box has a better fit around the geometry set (i.e., it is not assumed to be infinitely tall). Figure 14.79 reminds us what a Y-variant quad-tree node might look like with each of its children not filling the entire volume of the parent node.

Figure 14.79

Because the bounding boxes are now built to the fit the size of the geometry contained at each node along the vertical extent, we get nodes with smaller boxes that provide more accurate querying versus the vanilla quad-tree case.

We can imagine a situation for example where a spacecraft might be hovering in the skies above a cityscape. In the vanilla quad-tree case, the leaf in which the cityscape is contained would be infinitely tall (or as tall as the tallest thing in the scene) and as such, the spacecraft would be considered to be in the same leaf node as the cityscape even though it was far above it. This would lead to the cityscape being rendered even if the plane was so far above it that that the geometry of the cityscape was well below the frustum of the camera mounted on that craft. In the Y-variant quad-tree case, the leaf in which the cityscape is contained would only be as tall as the tallest building in that city, which would mean the spacecraft would be outside that leaf and it would not be rendered. Of course, the savings are a lot less obvious when we are compiling indoor scenes.

While an oct-tree is one obvious way to solve this problem, we can make a modification to the quad-tree implementation and derive a new class called CQuadTreeYV that will be built slightly differently. At each node, the bounding box's Y extents are not inherited from the root node, but are calculated based on the polygon data in that node. This will allow us to more effectively frustum cull leaves that might not be culled in the vanilla quad-tree implementation.

Although it might seem that the Y-variant quad-tree would also introduce similar savings in collision queries (which is technically true), we will not be able to do this with our implementation. You will recall that the CollectLeavesAABB used by our collision system is implemented in the vanilla quad-tree case to assume infinitely tall leaves by way of ignoring the Y component of the node's box during AABB/AABB intersection tests (the CollectLeavesRay method does the same). Although we could easily re-implement these in our Y-variant version of the quad-tree, this design does not cooperate well with our dynamic object system.

Looking at Figure 14.79 we can see that at any given node, if the child volumes do not totally consume the parent volume, we have a situation where locations within the quad-tree do not fall within any leaf

node. In the circular inset we can see some children have a very small Y extent while the Y extent of the parent node would be equal to the green child. What happens if we feed an AABB into the tree which happens to fall in the space just above the blue child node? It would clearly be contained inside the parent node's volume, but it would not fall into any of its children. Thus, it would essentially be in no-man's land and the function would simply return and no leaf would be added to the leaf collection list.

Now, if this was a collision query being performed, we would really benefit from the Y-variant tree's smaller boxes. If we passed the swept sphere's AABB down the tree and it ended up in an area that is not bounded by a leaf, the returned list would be empty and we would know that the swept sphere does not intersect any geometry. However, the same CollectLeavesAABB method is also used by our application to determine which leaves a dynamic object is in. The object is only rendered if at least one of its leaves is visible. However, in the Y-variant case we might find that when we feed in the dynamic object's bounding volume, it exists in no leaves. What do we do?

We would have no choice in such situations other than to always render those dynamic objects. That is, if a dynamic object is not currently in a leaf, we do not know whether or not it is visible, so we should render it to be safe. We could end up rendering many objects which are nowhere near the camera and not even close to being visible, and this would certain hinder performance in many cases. Of course, there are ways around such problems; we could store dynamic object pointers at nodes as well as leaves and when traversing the tree, render any objects that are contained at that node before moving on. However, that really does not fit our design very well, where we make a clear distinction between nodes and leaves. So we have made the design decision to ignore the y extents of a node's box during the leaf collection functions. This means that, just as in the case of the vanilla quad-tree, during the leaf collection process the node volumes are assumed to have infinite Y extents. The savings we get from implementing the YV quad-tree will be purely during the frustum culling pass of the tree for static objects.

Note that this tree will behave almost exactly like the vanilla quad-tree -- even the CollectLeavesAABB method will be the same since we are choosing to ignore the Y components. In fact, the only difference (frustum culling aside, as this will be discussed in the next lesson) between the quad-tree and the YV quad-tree will be a small section of code in the recursive BuildTree function that fits the bounding box of the node to the polygon set along the Y extents, instead of inheriting it from the root node's volume. As the BuildTree method of CQuadTree is virtual, we can simply derive CQuadTreeYV from CQuadTree and just override it.

Below we show the class declaration for CQuadTreeYV which is contained in CQuadTreeYV.h.

```
class CQuadTreeYV : public CQuadTree
{
public:

    // Constructors & Destructors for This Class.
    CQuadTreeYV( LPDIRECT3DDEVICE9 pDevice,
                 bool bHardwareTnL,
                 float fMinLeafSize = 300.0f,
                 ULONG nMinPolyCount = 600,
                 ULONG nMinAreaCount = 0 ) : CQuadTree( pDevice,
                                                        bHardwareTnL,
                                                        fMinLeafSize,
```

```
                                                    nMinPolyCount,
                                                    nMinAreaCount ) {};

protected:
    // Protected virtual Functions for This Class

    virtual bool BuildTree ( CQuadTreeNode * pNode,
                             PolygonList PolyList,
                             DetailAreaList AreaList,
                             const D3DXVECTOR3 & BoundsMin,
                             const D3DXVECTOR3 & BoundsMax );
};
```

As you can see, with the exception of the constructor which is empty and simply passes the data on to the base class (CQuadTree), we have to implement just one function. The BuildTree method is a method we are now very familiar with. We will override it in this class due to the fact that we need to calculate the bounding box of each node a little differently. We will now cover the BuildTree method and highlight the small differences in the YV version.

BuildTree – CQuadTreeYV

Although we will see the complete function code listing below, most of it is unchanged from the CQuadTree::BuildTree method. Recall that this is the method that is called from the Build method and passed the root node and its polygon and detail area data along with the bounding box of the root node. This function will then recursively build the entire tree.

Because this code has already been thoroughly discussed, we will move quickly and only stop to examine the changes from the function that it overrides.

```
bool CQuadTreeYV::BuildTree( CQuadTreeNode * pNode,
                             PolygonList PolyList,
                             DetailAreaList AreaList,
                             const D3DXVECTOR3 & BoundsMin,
                             const D3DXVECTOR3 & BoundsMax )
{
    D3DXVECTOR3                 Normal;
    PolygonList::iterator       PolyIterator;
    DetailAreaList::iterator    AreaIterator;
    CPolygon                  * CurrentPoly, * FrontSplit, * BackSplit;
    PolygonList                 ChildList[4];
    DetailAreaList              ChildAreaList[4];
    CCollision::CLASSIFYTYPE    Location[2];
    D3DXPLANE                   Planes[2];
    unsigned long               i;
    bool                        bStopCode;

    // Store the bounding box properties in the node
    pNode->BoundsMin = BoundsMin;
    pNode->BoundsMax = BoundsMax;
```

```
    // Calculate 'Stop' code
    D3DXVECTOR3 vecLeafSize = D3DXVECTOR3( BoundsMax.x - BoundsMin.x,
                                           0,
                                           BoundsMax.z - BoundsMin.z );

    bStopCode  = (AreaList.size() == 0 && PolyList.size() == 0) ||
                 (AreaList.size() <= m_nMinAreaCount &&
                  PolyList.size() <= m_nMinPolyCount) ||
                  D3DXVec3Length( &vecLeafSize ) <= m_fMinLeafSize;
```

The first section code stored the passed bounding box in the passed node and calculated the 2D size of the node (vecLeafSize). Although this is a YV quad-tree we still do not want the y extents of the node to play any part in whether or not we subdivide. Unlike the oct-tree where we intend to subdivide space vertically, the YV quad-tree does not have leaves stacked on top of each other. So a leaf must still contain the entire vertical range of geometry that exists in the XZ space. The only difference is that when we do create a node/leaf bounding box, we will calculate the extents of the actual geometry that made it into that node. After we have calculated the leaf size, we compute the bStopCode boolean. This code is all unchanged.

As before, if the stop code is set to true then we have found a node that should become a leaf so we allocate a new CBaseLeaf object. We then loop through each polygon in the passed list (the polygons that made it into this node) and add their pointers to the leaf. Remember that if this is a clipped tree, we also add the polygons to the tree's polygon list.

```
    // If we reached our stop limit, build a leaf here
    if ( bStopCode )
    {
        // Build a leaf
        CBaseLeaf * pLeaf = new CBaseLeaf( this );
        if ( !pLeaf ) return false;

        // Store the polygons
        for ( PolyIterator = PolyList.begin();
              PolyIterator != PolyList.end();
              ++PolyIterator )
        {
            // Retrieve poly
            CurrentPoly = *PolyIterator;
            if ( !CurrentPoly ) continue;

            // Add to the polygon list ONLY if splitting was allowed
            if ( m_bAllowSplits ) AddPolygon( CurrentPoly );

            // Also add a reference to the leaf's list
            pLeaf->AddPolygon( CurrentPoly );

        } // Next Polygon
```

Of course, we do the same for any detail area that made it into this leaf.

```
        // Store the area lists
        for ( AreaIterator = AreaList.begin();
```

```
              AreaIterator != AreaList.end(); ++AreaIterator )
        {
            // Retrieve detail area item
            TreeDetailArea * pDetailArea = *AreaIterator;
            if ( !pDetailArea ) continue;

            // Add a reference to the leaf's list
            pLeaf->AddDetailArea( pDetailArea );

        } // Next Polygon
```

We then assign the node's leaf pointer to point at the newly populated CBaseLeaf and set the leaf object's bounding box to that of the node it represents. We then add the leaf pointer to the tree's leaf array and return. That takes care of the leaf case.

```
        // Store pointer to leaf in the node and the bounds
        pNode->Leaf = pLeaf;

        // Store the bounding box in the leaf
        pLeaf->SetBoundingBox( pNode->BoundsMin, pNode->BoundsMax );

        // Store the leaf in the leaf list
        AddLeaf( pLeaf );

        // We have reached a leaf
        return true;

    } // End if reached stop code
```

The next piece of code is new for the Y-variant quad-tree. Remember that we only reach this part of the function if the current node is not a leaf. Although we already know the X and Z extents of the node's bounding box because they were passed into the function, the y extents will currently represent the y extents of the parent node's volume. We want to set the height of this node's volume so that it matches the y range of vertices it stores. Therefore, we first set the y extents of the node's AABB to impossibly small (inside out) values. We then loop through every polygon in the passed list (the polygons contained in this node) and test the y extents of each of its vertices against the y extents of the node's box. Whenever we find a vertex that has a y component that is outside the box, we grow the box extents to contain it. At the end of the following section of code, the current node's bounding box will be an exact fit (vertically) around the polygon data passed into this node.

```
    // Update the Y coordinate of the bounding box for the 'Y Variant'
    // quad-tree information

    pNode->BoundsMin.y = FLT_MAX; pNode->BoundsMax.y = -FLT_MAX;

    for ( PolyIterator = PolyList.begin();
          PolyIterator != PolyList.end(); ++PolyIterator )
    {
        // Store current poly
        CurrentPoly = *PolyIterator;
        if ( !CurrentPoly ) continue;
```

```
        // Loop through each vertex
        for ( i = 0; i < CurrentPoly->m_nVertexCount; ++i )
        {
            CVertex * pVertex = &CurrentPoly->m_pVertex[i];
            if ( pVertex->y < pNode->BoundsMin.y ) pNode->BoundsMin.y = pVertex->y;
            if ( pVertex->y > pNode->BoundsMax.y ) pNode->BoundsMax.y = pVertex->y;

        } // Next Vertex

    } // Next Polygon
```

Of course, we must make sure that the box is also large enough to contain any detail areas that made it into this node. In the next section of code we loop through each detail area in this node and grow the y extents of the node's AABB if any detail area is found to not be fully contained.

```
    for ( AreaIterator = AreaList.begin();
          AreaIterator != AreaList.end(); ++AreaIterator )
    {
        // Store current detail area
        TreeDetailArea * pDetailArea = *AreaIterator;

        if ( pDetailArea->BoundsMin.x < pNode->BoundsMin.x )
             pNode->BoundsMin.x = pDetailArea->BoundsMin.x;

        if ( pDetailArea->BoundsMin.y < pNode->BoundsMin.y )
             pNode->BoundsMin.y = pDetailArea->BoundsMin.y;

        if ( pDetailArea->BoundsMin.z < pNode->BoundsMin.z )
             pNode->BoundsMin.z = pDetailArea->BoundsMin.z;

        if ( pDetailArea->BoundsMax.x > pNode->BoundsMax.x )
             pNode->BoundsMax.x = pDetailArea->BoundsMax.x;

        if ( pDetailArea->BoundsMax.y > pNode->BoundsMax.y )
             pNode->BoundsMax.y = pDetailArea->BoundsMax.y;

        if ( pDetailArea->BoundsMax.z > pNode->BoundsMax.z )
             pNode->BoundsMax.z = pDetailArea->BoundsMax.z;

    } // Next Detail Area
```

With the node's bounding volume now a tight fit around its data, we create the two split planes as before.

```
    // Generate the two split planes
    D3DXPlaneFromPointNormal( &Planes[0],
                              &((pNode->BoundsMin + pNode->BoundsMax) / 2.0f),
                              &D3DXVECTOR3( 0.0f, 0.0f, 1.0f ) );

    D3DXPlaneFromPointNormal( &Planes[1],
                              &((pNode->BoundsMin + pNode->BoundsMax) / 2.0f),
                              &D3DXVECTOR3( 1.0f, 0.0f, 0.0f ) );
```

If we are building a clipped tree then we will first need to split all the polygons in this node's list to the planes so that we end up with a list of polygons that will each fit into exactly one child.

```
    // Split all polygons against both planes if required
    if ( m_bAllowSplits )
    {
        for ( i = 0; i < 2; ++i )
        {
            for ( PolyIterator = PolyList.begin();
                  PolyIterator != PolyList.end(); ++PolyIterator )
            {
                // Store current poly
                CurrentPoly = *PolyIterator;
                if ( !CurrentPoly ) continue;

        // Classify the poly against the first plane
        Location[0] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                     CurrentPoly->m_nVertexCount,
                                                     sizeof(CVertex),
                                                     (D3DXVECTOR3&)Planes[i],
                                                     Planes[i].d );

                if ( Location[0] == CCollision::CLASSIFY_SPANNING )
                {
                    // Split the current poly against the plane
                    CurrentPoly->Split( Planes[i], &FrontSplit, &BackSplit );
                    delete CurrentPoly;
                    *PolyIterator = NULL;

                    // Add these to the end of the current poly list
                    PolyList.push_back( FrontSplit );
                    PolyList.push_back( BackSplit );

                } // End if Spanning

            } // Next Polygon

        } // Next Plane

    } // End if allow splits
```

As before, the next section of code adds each polygon to the appropriate child list. It is unchanged from the normal quad-tree case.

```
    // Classify the polygons and sort them into the four child lists.
    for ( PolyIterator = PolyList.begin();
          PolyIterator != PolyList.end(); ++PolyIterator )
    {
        // Store current poly
        CurrentPoly = *PolyIterator;
        if ( !CurrentPoly ) continue;

        // Classify the poly against the planes
        Location[0] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                     CurrentPoly->m_nVertexCount,
```

```
                                                    sizeof(CVertex),
                                                    (D3DXVECTOR3&)Planes[0],
                                                    Planes[0].d );

        Location[1] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                    CurrentPoly->m_nVertexCount,
                                                    sizeof(CVertex),
                                                    (D3DXVECTOR3&)Planes[1],
                                                    Planes[1].d );

        // Position relative to XY plane
        if ( Location[0] == CCollision::CLASSIFY_BEHIND ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                            ChildList[0].push_back( CurrentPoly );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_ONPLANE ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                             ChildList[1].push_back( CurrentPoly );

        } // End if behind or spanning

        if ( Location[0] == CCollision::CLASSIFY_INFRONT ||
             Location[0] == CCollision::CLASSIFY_ONPLANE ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                            ChildList[2].push_back( CurrentPoly );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_ONPLANE ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                            ChildList[3].push_back( CurrentPoly );

        } // End if in-front or on-plane

    } // Next Triangle
```

At this point, the polygon data at this node has been sorted into four lists (one per quadrant). We now have to sort the detail areas into four lists also so that we know which children they should be assigned to. This code is also unchanged from the normal quad-tree code.

```
    // Classify the areas and sort them into the child lists.
    for ( AreaIterator = AreaList.begin();
          AreaIterator != AreaList.end(); ++AreaIterator )
    {
        // Store current area
        TreeDetailArea * pDetailArea = *AreaIterator;
        if ( !pDetailArea ) continue;
```

```
        // Classify the poly against the planes
        Location[0] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)Planes[0],
                                                     Planes[0].d );

        Location[1] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)Planes[1],
                                                     Planes[1].d );

        // Position relative to XY plane
        if ( Location[0] == CCollision::CLASSIFY_BEHIND ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildAreaList[0].push_back( pDetailArea );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildAreaList[1].push_back( pDetailArea );

        } // End if behind or spanning

        if ( Location[0] == CCollision::CLASSIFY_INFRONT ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildAreaList[2].push_back( pDetailArea );

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
                                ChildAreaList[3].push_back( pDetailArea );

        } // End if in-front or on-plane

    } // Next Detail Area
```

We now have four lists of polygons and four lists of detail objects, so we create the four child nodes to which they belong and calculate the bounding box of each child. As before, we set up a loop where a single child node is created with each iteration. In each of the four iterations, we allocate a new quad-tree node and store its pointer in the parent node's child list and then calculate the bounding volume of that node by assigning the associated quadrant of the parent node's bounding volume. We then step into the child node by issuing the recursive call.

```
    // Build each of the children here
    for( i = 0; i < 4; ++i )
    {
        // Calculate child bounding box values
        D3DXVECTOR3 NewBoundsMin, NewBoundsMax,
```

```
                MidPoint = (pNode->BoundsMin + pNode->BoundsMax) / 2.0f;

        switch( i )
        {
        case 0: // Behind Left
            NewBoundsMin = pNode->BoundsMin;
            NewBoundsMax = D3DXVECTOR3( MidPoint.x,
                                        pNode->BoundsMax.y,
                                        MidPoint.z);
            break;

        case 1: // Behind Right
            NewBoundsMin = D3DXVECTOR3( MidPoint.x,
                                        pNode->BoundsMin.y,
                                        pNode->BoundsMin.z );

            NewBoundsMax = D3DXVECTOR3( pNode->BoundsMax.x,
                                        pNode->BoundsMax.y,
                                        MidPoint.z);
            break;


        case 2: // InFront Left
            NewBoundsMin = D3DXVECTOR3( pNode->BoundsMin.x,
                                        pNode->BoundsMin.y,
                                        MidPoint.z );

            NewBoundsMax = D3DXVECTOR3( MidPoint.x,
                                        pNode->BoundsMax.y,
                                        pNode->BoundsMax.z );
            break;

        case 3: // InFront Right
            NewBoundsMin = D3DXVECTOR3( MidPoint.x,
                                        pNode->BoundsMin.y,
                                        MidPoint.z );

            NewBoundsMax = pNode->BoundsMax;
            break;

        } // End Child Type Switch


        // Allocate child node
        pNode->Children[i] = new CQuadTreeNode;
        if ( !pNode->Children[i] ) return false;

        // Recurse into this new node
        BuildTree( pNode->Children[i],
                   ChildList[i],
                   ChildAreaList[i],
                   NewBoundsMin,
                   NewBoundsMax );

        // Clean up
        ChildList[i].clear();
        ChildAreaList[i].clear();
```

```
    } // Next Child

    // Success!
    return true;
}
```

Note how quickly we were able to create new tree behavior. In the case of the Y-variant quad-tree, it meant overriding a single method. That is all the code we need to cover for the YV Quad-tree.

14.19 Oct-Tree Implementation

The source files COctTree.cpp and COctTree.h contain the code for our oct-tree implementation. This tree is derived from CBaseTree and as such, we must implement the same methods from ISpatialTree that are not implemented in CBaseTree. Just as with the quad-tree, we will only need to implement the Build, CollectLeavesAABB, CollectLeavesRay and a few others in order to create our oct-tree because CBaseTree provides all the housekeeping tasks.

The oct-tree implementation will be so similar to that of the quad-tree that examining its code will be trivial for the most part. The only real difference in each version of the method implemented for the oct-tree is that each node has eight children to visit/create instead of four. In the Build function we will now have three planes to clip and classify against instead of two. These three planes will divide the node's volume in octants (instead of quadrants) as shown in Figure 14.80.

Octree Node Partitioning

Figure 14.80

Although CBaseTree provides the leaf structure that all our trees will use, the node structure is dependant on the tree being built. For example, in the quad-tree case we implemented the CQuadTreeNode object to represent a single node in the tree. This node has four child pointers as we would expect. The Y-variant quad-tree also shared this same node structure. The oct-tree will need its own node structure that is capable or storing pointers to eight child nodes instead of four. That is really the only difference between the node objects used by all of our tree types; the varying number of children.

14.19.1 COctTreeNode – The Source Code

Each node in our oct-tree will be represented by an object of type COctTreeNode which is declared and implemented in COctTree.h and COctTree.cpp. It contains the same SetVisible member function as its quad-tree counterpart and also stores the same members. It stores a pointer to a CBaseLeaf object that will be used only by terminal nodes and contains two vectors describing the extents of the node's bounding volume. It also contains the same LastFrustumPlane member which we will discuss in the following lesson as it is used to speed up visibility processing. Finally, you will notice that the only difference (except the name) between this object and the CQuadTreeNode object is that each node stores an array of 8 COctTreeNode pointers instead of an array of 4 CQuadTreeNode pointers.

```
class COctTreeNode
{
public:

    // Constructors & Destructors for This Class.
     COctTreeNode( );
    ~COctTreeNode( );


    // Public Functions for This Class
    void SetVisible( bool bVisible );


    // Public Variables for This Class
    COctTreeNode  * Children[8];        // The eight child nodes
    CBaseLeaf     * Leaf;               // If this is a leaf, store here.
    D3DXVECTOR3     BoundsMin;          // Minimum bounding box extents
    D3DXVECTOR3     BoundsMax;          // Maximum bounding box extents
    signed char     LastFrustumPlane;   // 'last plane' index.
};
```

SetVisible – COctTreeNode

Each of our node types supports the SetVisible method which is used by the tree to set the visibility status of all leaves below the node for which it is called. It works in exactly the same way as its quad-tree node counterpart. How this function is used will become clearer in the following lesson when we implement the rendering and visibility code for our trees.

The method is passed a boolean parameter indicating whether the node is considered visible or not. If the node is a leaf then the attached leaf object has its visibility status set to true. If this is not a leaf we simply traverse into the eight children performing the same test. As a node has no member to store its visibility status, what we are actually doing here is just traversing down to find any leaf nodes and setting their status to the parameter passed to the top level instance of the function.

```
void COctTreeNode::SetVisible( bool bVisible )
{
    unsigned long i;

    // Set leaf property
    if ( Leaf  ) { Leaf->SetVisible( bVisible ); return; }

    // Recurse down if applicable
    for ( i = 0; i < 8; ++i )
    {
        if ( Children[i] ) Children[i]->SetVisible(bVisible);

    } // Next Child
}
```

Notice that the only difference between this code and the quad-tree node's method is that we now have a loop that visits eight children instead of four.

14.19.2 COctTree – The Source Code

The COctTree class is derived from CBaseTree and therefore must implement oct-tree versions of the methods we implemented in CQuadTree. The class declaration is contained in COctTree.h and is shown below. Notice that it implements the exact same set of methods we had to implement for our quad-tree class.

```
class COctTree : public CBaseTree
{
public:

       // Constructors & Destructors for This Class.
       virtual ~COctTree( );
       COctTree( LPDIRECT3DDEVICE9 pDevice,
                 bool bHardwareTnL,
                 float fMinLeafSize = 300.0f,
                 ULONG nMinPolyCount = 600,
                 ULONG nMinAreaCount = 0 );


    // Public Virtual Functions for This Class (from base).
    virtual bool Build                  ( bool bAllowSplits = true );
    virtual void ProcessVisibility      ( CCamera & Camera );
    virtual bool CollectLeavesAABB      ( LeafList & List,
                                          const D3DXVECTOR3 & Min,
                                          const D3DXVECTOR3 & Max );
    virtual bool CollectLeavesRay       ( LeafList & List,
                                          const D3DXVECTOR3 & RayOrigin,
                                          const D3DXVECTOR3 & Velocity );

    virtual void DebugDraw              ( CCamera & Camera );
    virtual bool GetSceneBounds         ( D3DXVECTOR3 & Min, D3DXVECTOR3 & Max );
```

```
protected:

    // Protected virtual Functions for This Class
    virtual bool BuildTree              ( COctTreeNode * pNode,
                                          PolygonList PolyList,
                                          DetailAreaList AreaList,
                                          const D3DXVECTOR3 & BoundsMin,
                                          const D3DXVECTOR3 & BoundsMax );

    // Protected Functions for This Class
    void         UpdateTreeVisibility   ( COctTreeNode * pNode,
                                          CCamera & Camera,
                                          UCHAR FrustumBits = 0x0 );

    bool         DebugDrawRecurse       ( COctTreeNode * pNode,
                                          CCamera & Camera, bool bRenderInLeaf );

    bool         CollectAABBRecurse     ( COctTreeNode * pNode,
                                          LeafList & List,
                                          const D3DXVECTOR3 & Min,
                                          const D3DXVECTOR3 & Max,
                                          bool bAutoCollect = false );

    bool         CollectRayRecurse      ( COctTreeNode * pNode,
                                          LeafList & List,
                                          const D3DXVECTOR3 & RayOrigin,
                                          const D3DXVECTOR3 & Velocity );

    // Protected Variables for This Class
    COctTreeNode * m_pRootNode;         // The root node of the tree
    bool           m_bAllowSplits;      // Is splitting allowed?
    float          m_fMinLeafSize;      // Min leaf size stop code
    ULONG          m_nMinPolyCount;     // Min polygon count stop code
    ULONG          m_nMinAreaCount;     // Min detail area count stop code

};
```

The member variables are exactly the same in both name and purpose to those we discussed when implementing the quad-tree. The only slight difference in member variables is that the root node pointer stored by the tree is now of type COctTreeNode instead of CQuadTreeNode. The methods are also exactly the same as their quad-tree counterparts with the exception that the recursive methods that accept node parameters now accept nodes of type COctTreeNode.

Let us now cover the methods of COctTree and discuss how they differ from their quad-tree counterparts.

Build - COctTree

As we know, the Build method is the method called by the application to build the tree. It is called after all geometry and detail areas have been registered. It is not the recursive function that builds the tree but

is instead a convenient wrapper around the top level call to the recursive BuildTree method. Because the code is almost unchanged from its quad-tree counter part we will progress through it quickly.

The first bit of the function initializes the vectors that will be used to record the size of the root node's bounding box to impossibly small values before allocating a new COctTreeNode object that will become the root node of the tree. We also store the value of the boolean parameter in the m_bAllowSplits member variable so that we know during the recursive build process whether we should be clipping the polygon data at each node.

```
bool COctTree::Build( bool bAllowSplits /* = false */ )
{
    PolygonList::iterator       PolyIterator = m_Polygons.begin();
    DetailAreaList::iterator    AreaIterator = m_DetailAreas.begin();
    PolygonList                 PolyList;
    DetailAreaList              AreaList;
    unsigned long               i;

    // Reset our tree info values.
    D3DXVECTOR3 vecBoundsMin( FLT_MAX, FLT_MAX, FLT_MAX );
    D3DXVECTOR3 vecBoundsMax( -FLT_MAX, -FLT_MAX, -FLT_MAX );

    // Allocate a new root node
    m_pRootNode = new COctTreeNode;
    if ( !m_pRootNode ) return false;

    // Store the allow splits value for later retrieval.
    m_bAllowSplits = bAllowSplits;
```

Now we need to calculate the size of the root nodes bounding volume by looping through every polygon in the tree's polygon list and testing each of its vertices against the current box extents. If we find any vertex is outside these extents, we grow the box. At the end of the loop we will have a bounding box that contains all the vertices of all the polygons.

```
    // Loop through all of the initial polygons
    for ( ; PolyIterator != m_Polygons.end(); ++PolyIterator )
    {
        // Retrieve the polygon
        CPolygon * pPoly = *PolyIterator;
        if ( !pPoly ) continue;

        // Calculate total scene bounding box.
        for ( i = 0; i < pPoly->m_nVertexCount; ++i )
        {
            // Store info
            CVertex * pVertex = &pPoly->m_pVertex[i];
            if ( pVertex->x < vecBoundsMin.x ) vecBoundsMin.x = pVertex->x;
            if ( pVertex->y < vecBoundsMin.y ) vecBoundsMin.y = pVertex->y;
            if ( pVertex->z < vecBoundsMin.z ) vecBoundsMin.z = pVertex->z;
            if ( pVertex->x > vecBoundsMax.x ) vecBoundsMax.x = pVertex->x;
            if ( pVertex->y > vecBoundsMax.y ) vecBoundsMax.y = pVertex->y;
            if ( pVertex->z > vecBoundsMax.z ) vecBoundsMax.z = pVertex->z;

        } // Next Vertex
```

```
        // Store this polygon in the top polygon list
        PolyList.push_back( pPoly );

    } // Next Polygon

    // Clear the initial polygon list if we are going to split the polygons
    // as this will eventually become storage for whatever gets built
    if ( bAllowSplits ) m_Polygons.clear();
```

Just as before, at the bottom of the outer loop, as we test each polygon, we add it to the temporary PolyList. It is this list that will be passed into the recursive process. As we are currently at the root node, PolyList will contain a complete copy of all the pointers that were registered with the tree and stored in the m_Polygons list. Outside the loop you can see that just as before, if we are going to build a clipped tree, we clear the tree's polygon list since we will need to rebuild it from scratch with the clipped polygon data that makes it into each leaf node.

Our root node's bounding box is currently large enough to contain all the polygon data but we need to make sure it is large enough to contain any detail areas that may have been registered with the tree as well. In the next section of code we loop through each detail area and extend the bounding box extents if any detail area is found not to be fully contained inside it. Notice also that as we process each detail area, we also add its pointer to the temporary detail area list. This is the detail area list we will send into the recursive build process.

```
    // Loop through all of the detail areas
    for ( ; AreaIterator != m_DetailAreas.end(); ++AreaIterator )
    {
        // Retrieve the detail area
        TreeDetailArea * pDetailArea = *AreaIterator;
        if ( !pDetailArea ) continue;

        // Calculate total scene bounding box.
        D3DXVECTOR3 & Min = pDetailArea->BoundsMin;
        D3DXVECTOR3 & Max = pDetailArea->BoundsMax;
        if ( Min.x < vecBoundsMin.x ) vecBoundsMin.x = Min.x;
        if ( Min.y < vecBoundsMin.y ) vecBoundsMin.y = Min.y;
        if ( Min.z < vecBoundsMin.z ) vecBoundsMin.z = Min.z;
        if ( Max.x > vecBoundsMax.x ) vecBoundsMax.x = Max.x;
        if ( Max.y > vecBoundsMax.y ) vecBoundsMax.y = Max.y;
        if ( Max.z > vecBoundsMax.z ) vecBoundsMax.z = Max.z;

        // Store this in the top detail area list
        AreaList.push_back( pDetailArea );

    } // Next Polygon
```

Finally, with the root node's bounding volume calculated and with PolyList and AreaList containing copies of all the polygon and detail area pointers, we can now step into the recursive BuildTree method starting at the root node and let it build the entire tree from the root node down.

```
    // Build the tree itself
    if (!BuildTree( m_pRootNode, PolyList, AreaList, vecBoundsMin, vecBoundsMax ))
```

```
        return false;
```

At this point the tree will be completely built so we call the CBaseTree::PostBuild method and let it do any final preparation on the tree data. We will see in the next lesson how this method not only calculates and stores the bounding boxes for each polygon stored in the tree (as we have already shown) but also initializes CBaseTree's render system.

```
    // Allow our base class to finish any remaining processing
    return CBaseTree::PostBuild( );
}
```

BuildTree - COctTree

The BuildTree method has the task of dividing the current node's polygon and detail area lists into eight children instead of four and creating eight child nodes. However, with the exception that we must now carve the node into octants using three split planes and creating eight child lists of both polygon and detail area data, the function is essentially the same as the quad-tree version.

Notice in this first section of code how we now instantiate local arrays of eight polygon lists and eight detail area lists and also allocate an array to hold three split planes instead of two.

```
bool COctTree::BuildTree( COctTreeNode * pNode,
                          PolygonList PolyList,
                          DetailAreaList AreaList,
                          const D3DXVECTOR3 & BoundsMin,
                          const D3DXVECTOR3 & BoundsMax )
{
    D3DXVECTOR3                 Normal;
    PolygonList::iterator       PolyIterator;
    DetailAreaList::iterator    AreaIterator;
    CPolygon                  * CurrentPoly, * FrontSplit, * BackSplit;
    PolygonList                 ChildList[8];
    DetailAreaList              ChildAreaList[8];
    CCollision::CLASSIFYTYPE    Location[3];
    D3DXPLANE                   Planes[3];
    unsigned long               i;
    bool                        bStopCode;

    // Store the bounding box properties in the node
    pNode->BoundsMin = BoundsMin;
    pNode->BoundsMax = BoundsMax;

    // Calculate 'Stop' code
    D3DXVECTOR3 vecLeafSize = BoundsMax - BoundsMin;

    bStopCode  = (AreaList.size() == 0 && PolyList.size() == 0) ||
                 (AreaList.size() <= m_nMinAreaCount &&
                  PolyList.size() <= m_nMinPolyCount ) ||
                  D3DXVec3Length( &vecLeafSize ) <= m_fMinLeafSize;
```

Just as before we store the passed bounding box extents in the node and then calculate the leaf size. Notice that this time when calculating the leaf size variable; we also take the y extents into account. Remember, in the quad-tree case we only subtracted the x and z extents of the bounding box to get the diagonal node length. This was because in a quad-tree, we do not partition space vertically. However, in the case of the oct-tree, we take the full diagonal length of the box into account so that the height of a node's bounding box is also a factor in whether or not it becomes a terminal node. As you can see, by simply subtracting the minimum box extents from the maximum box extents, we get a vector whose length describes the minimum size a node can be before it automatically becomes a leaf.

If the stop code variable is true, it means there are two few polygons or detail areas in this node or that the node is too small. Either way, we no longer wish to further partition the space of this node and instead wish to make it a leaf. The following conditional code that makes the node a leaf is completely unchanged from the quad-tree case. It creates a new CBaseLeaf and loops through every polygon and detail area in the node's list and adds them to the leaf. If clipping is being used it is at this point that the pointer of each potentially clipped polygon is also added back to the tree's main polygon list. The node's leaf pointer is then assigned to point at this new leaf and the leaf's bounding box is set to be the same as the node. At this point the node has been successfully turned into a leaf and we return.

```
    // If we reached our stop limit, build a leaf here
    if ( bStopCode )
    {
        // Build a leaf
        CBaseLeaf * pLeaf = new CBaseLeaf( this );
        if ( !pLeaf ) return false;

        // Store the polygons
        for ( PolyIterator = PolyList.begin();
              PolyIterator != PolyList.end(); ++PolyIterator )
        {
            // Retrieve poly
            CurrentPoly = *PolyIterator;
            if ( !CurrentPoly ) continue;

            // Add to full tree polygon list ONLY if splitting was allowed
            if ( m_bAllowSplits ) AddPolygon( CurrentPoly );

            // Also add a reference to the leaf's list
            pLeaf->AddPolygon( CurrentPoly );

        } // Next Polygon

        // Store the area lists
        for ( AreaIterator = AreaList.begin();
              AreaIterator != AreaList.end(); ++AreaIterator )
        {
            // Retrieve detail area item
            TreeDetailArea * pDetailArea = *AreaIterator;
            if ( !pDetailArea ) continue;

            // Add a reference to the leaf's list
            pLeaf->AddDetailArea( pDetailArea );

        } // Next Polygon
```

```
        // Store pointer to leaf in the node and the bounds
        pNode->Leaf = pLeaf;

        // Store the bounding box in the leaf
        pLeaf->SetBoundingBox( pNode->BoundsMin, pNode->BoundsMax );

        // Store the leaf in the leaf list
        AddLeaf( pLeaf );

        // We have reached a leaf
        return true;

    } // End if reached stop code
```

If we make it this far in the function then it means the node is not a leaf and we will further divide its space and its data into eight children. To divide an oct-tree node, we create three planes with which to split and assign the polygon and detail area data to each child list. The first two planes are identical to the planes of the quad-tree node (a plane facing down the positive Z axis and a plane facing down the positive X axis). Now we add a third plane whose normal faces along the positive Y axis and splits space vertically. All three planes intersect at the center of the node and therefore the point on plane used to create each plane is simply the center point of the box.

```
    // Generate the three split planes
    D3DXPlaneFromPointNormal( &Planes[0],
                              &((BoundsMin + BoundsMax) / 2.0f),
                              &D3DXVECTOR3( 0.0f, 0.0f, 1.0f ) );

    D3DXPlaneFromPointNormal( &Planes[1],
                              &((BoundsMin + BoundsMax) / 2.0f),
                              &D3DXVECTOR3( 1.0f, 0.0f, 0.0f ) );

    D3DXPlaneFromPointNormal( &Planes[2],
                              &((BoundsMin + BoundsMax) / 2.0f),
                              &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
```

If we are building a clipped tree then we will need to loop through each of these three planes, and for each plane we will need to classify every polygon in the node's list against it. If we find any polygon spanning a plane, it is split by the plane, the original polygon is deleted from the list, and the two new split polygons are added. This code is exactly the same as the previous versions of the function we have seen with the exception that the outer loop now iterates through three planes instead of two.

```
    // Split all polygons against all three planes if required
    if ( m_bAllowSplits )
    {
        for ( i = 0; i < 3; ++i )
        {
            for ( PolyIterator = PolyList.begin();
                  PolyIterator != PolyList.end(); ++PolyIterator )
            {
                // Store current poly
                CurrentPoly = *PolyIterator;
                if ( !CurrentPoly ) continue;
```

```
        // Classify the poly against the first plane
        Location[0] = CCollision::PolyClassifyPlane(CurrentPoly->m_pVertex,
                                                    CurrentPoly->m_nVertexCount,
                                                    sizeof(CVertex),
                                                    (D3DXVECTOR3&)Planes[i],
                                                    Planes[i].d );

             if ( Location[0] == CCollision::CLASSIFY_SPANNING )
             {
                 // Split the current poly against the plane
                 CurrentPoly->Split( Planes[i], &FrontSplit, &BackSplit );

                 delete CurrentPoly;
                 *PolyIterator = NULL;

                 // Add these to the end of the current poly list
                 PolyList.push_back( FrontSplit );
                 PolyList.push_back( BackSplit );

             } // End if Spanning

         } // Next Polygon

     } // Next Plane

 } // End if allow splits
```

At this point if clipping was desired all the polygons in the node's list will have been clipped and will fit neatly into one of the octants of the node. Our task is to loop through every polygon in the list and find out in which octant it is contained. This is done by classifying the polygon against the three planes and analyzing the result. For example, we know that if it is behind the first plane it must lay in the back halfspace of the node. If it is in front of the second plane we know that it is somewhere in the back right area of the node. Finally, if it is in front of the third plane we know that it is contained in the upper back right octant of the node. The strategy in this code is exactly the same as in the quad-tree version except now we have an extra plane to classify each polygon against, adding a few more conditionals.

```
 // Classify the polygons and sort them into the child lists.
 for ( PolyIterator = PolyList.begin();
       PolyIterator != PolyList.end(); ++PolyIterator )
 {
     // Store current poly
     CurrentPoly = *PolyIterator;
     if ( !CurrentPoly ) continue;

     // Classify the poly against the planes
     Location[0] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                  CurrentPoly->m_nVertexCount,
                                                  sizeof(CVertex),
                                                  (D3DXVECTOR3&)Planes[0],
                                                  Planes[0].d );

     Location[1] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                  CurrentPoly->m_nVertexCount,
```

```
                                                    sizeof(CVertex),
                                                    (D3DXVECTOR3&)Planes[1],
                                                    Planes[1].d );

        Location[2] = CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                    CurrentPoly->m_nVertexCount,
                                                    sizeof(CVertex),
                                                    (D3DXVECTOR3&)Planes[2],
                                                    Planes[2].d );
```

With the classification results for the current polygon stored in the three element Location array, let us now analyze the three results and work out in which octants the polygon is contained and to which child lists the polygon's pointers should be added.

```
        // Position relative to XY plane
        if ( Location[0] == CCollision::CLASSIFY_BEHIND ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
            {
                // Position relative to XZ plane
                if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[0].push_back( CurrentPoly );

                if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                     Location[2] == CCollision::CLASSIFY_ONPLANE ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                     ChildList[4].push_back( CurrentPoly );

            } // End if behind

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_ONPLANE ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
            {
                // Position relative to XZ plane
                if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[1].push_back( CurrentPoly );

                if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                     Location[2] == CCollision::CLASSIFY_ONPLANE ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[5].push_back( CurrentPoly );

            } // End if in-front or on-plane

        } // End if behind

        if ( Location[0] == CCollision::CLASSIFY_INFRONT ||
             Location[0] == CCollision::CLASSIFY_ONPLANE ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
```

```
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
            {
                // Position relative to XZ plane
                if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[2].push_back( CurrentPoly );

                if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                     Location[2] == CCollision::CLASSIFY_ONPLANE ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[6].push_back( CurrentPoly );

            } // End if behind

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_ONPLANE ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
            {
                // Position relative to XZ plane
                if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[3].push_back( CurrentPoly );

                if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                     Location[2] == CCollision::CLASSIFY_ONPLANE ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildList[7].push_back( CurrentPoly );

            } // End if in-front or on-plane

        } // End if in-front or on-plane

    } // Next Triangle
```

At this point we have added all the polygon pointers for this node to the child lists so we must also do the same for the detail areas. The next section of code classifies the detail areas of this node against the three node planes to find out which child lists the detail areas should be added to. The code is almost the same as the quad-tree version with the exception that we now have an extra plane to classify against and more results to analyze.

```
    // Classify the areas and sort them into the child lists.
    for ( AreaIterator = AreaList.begin();
          AreaIterator != AreaList.end(); ++AreaIterator )
    {
        // Store current area
        TreeDetailArea * pDetailArea = *AreaIterator;
        if ( !pDetailArea ) continue;

        // Classify the area against the planes
        Location[0] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)Planes[0],
```

```
                                                Planes[0].d );

        Location[1] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)Planes[1],
                                                     Planes[1].d );

        Location[2] = CCollision::AABBClassifyPlane( pDetailArea->BoundsMin,
                                                     pDetailArea->BoundsMax,
                                                     (D3DXVECTOR3&)Planes[2],
                                                     Planes[2].d );

        // Position relative to XY plane
        if ( Location[0] == CCollision::CLASSIFY_BEHIND ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
            {
                // Position relative to XZ plane
                if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildAreaList[0].push_back( pDetailArea );

                if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildAreaList[4].push_back( pDetailArea );

            } // End if behind

            if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
            {
                // Position relative to XZ plane
                if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildAreaList[1].push_back( pDetailArea );

                if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                     Location[2] == CCollision::CLASSIFY_SPANNING)
                                    ChildAreaList[5].push_back( pDetailArea );

            } // End if in-front or on-plane

        } // End if behind


        if ( Location[0] == CCollision::CLASSIFY_INFRONT ||
             Location[0] == CCollision::CLASSIFY_SPANNING )
        {
            // Position relative to ZY plane
            if ( Location[1] == CCollision::CLASSIFY_BEHIND ||
                 Location[1] == CCollision::CLASSIFY_SPANNING )
            {
                // Position relative to XZ plane
                if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
```

```
                    Location[2] == CCollision::CLASSIFY_SPANNING)
                                   ChildAreaList[2].push_back( pDetailArea );

               if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                    Location[2] == CCollision::CLASSIFY_SPANNING)
                                   ChildAreaList[6].push_back( pDetailArea );

           } // End if behind

           if ( Location[1] == CCollision::CLASSIFY_INFRONT ||
                Location[1] == CCollision::CLASSIFY_SPANNING )
           {
               // Position relative to XZ plane
               if ( Location[2] == CCollision::CLASSIFY_BEHIND ||
                    Location[2] == CCollision::CLASSIFY_SPANNING)
                                   ChildAreaList[3].push_back( pDetailArea );

               if ( Location[2] == CCollision::CLASSIFY_INFRONT ||
                    Location[2] == CCollision::CLASSIFY_SPANNING)
                                   ChildAreaList[7].push_back( pDetailArea );

           } // End if in-front or on-plane

       } // End if in-front or on-plane

   } // Next Detail Area
```

At this point we have the eight polygon lists and the eight detail area lists, so it is time to create each child node, compute its bounding box, and store a pointer to each child node in the parent node's child list.

```
   // Build each of the children here
   for( i = 0; i < 8; ++i )
   {
       // Calculate child bounding box values
       D3DXVECTOR3 NewBoundsMin, NewBoundsMax,
                   MidPoint = (BoundsMin + BoundsMax) / 2.0f;
       switch( i )
       {
           case 0: // Bottom Behind Left
               NewBoundsMin = BoundsMin;
               NewBoundsMax = D3DXVECTOR3( MidPoint.x, MidPoint.y, MidPoint.z);
               break;

           case 1: // Bottom Behind Right
               NewBoundsMin = D3DXVECTOR3( MidPoint.x, BoundsMin.y, BoundsMin.z );
               NewBoundsMax = D3DXVECTOR3( BoundsMax.x, MidPoint.y, MidPoint.z);
               break;

           case 2: // Bottom InFront Left
               NewBoundsMin = D3DXVECTOR3( BoundsMin.x, BoundsMin.y, MidPoint.z );
               NewBoundsMax = D3DXVECTOR3( MidPoint.x, MidPoint.y, BoundsMax.z );
               break;

           case 3: // Bottom InFront Right
```

```
                NewBoundsMin = D3DXVECTOR3( MidPoint.x, BoundsMin.y, MidPoint.z );
                NewBoundsMax = D3DXVECTOR3( BoundsMax.x, MidPoint.y, BoundsMax.z );
                break;

            case 4: // Top Behind Left
                NewBoundsMin = D3DXVECTOR3( BoundsMin.x, MidPoint.y, BoundsMin.z );
                NewBoundsMax = D3DXVECTOR3( MidPoint.x, BoundsMax.y, MidPoint.z);
                break;

            case 5: // Top Behind Right
                NewBoundsMin = D3DXVECTOR3( MidPoint.x, MidPoint.y, BoundsMin.z );
                NewBoundsMax = D3DXVECTOR3( BoundsMax.x, BoundsMax.y, MidPoint.z);
                break;

            case 6: // Top InFront Left
                NewBoundsMin = D3DXVECTOR3( BoundsMin.x, MidPoint.y, MidPoint.z );
                NewBoundsMax = D3DXVECTOR3( MidPoint.x, BoundsMax.y, BoundsMax.z );
                break;

            case 7: // Top InFront Right
                NewBoundsMin = D3DXVECTOR3( MidPoint.x, MidPoint.y, MidPoint.z );
                NewBoundsMax = BoundsMax;
                break;

        } // End Child Type Switch

        // Allocate child node
        pNode->Children[i] = new COctTreeNode;
        if ( !pNode->Children[i] ) return false;

        // Recurse into this new node
        BuildTree( pNode->Children[i],
                   ChildList[i],
                   ChildAreaList[i],
                   NewBoundsMin,
                   NewBoundsMax );

        // Clean up
        ChildList[i].clear();
        ChildAreaList[i].clear();

    } // Next Child

    // Success!
    return true;
}
```

At the end of the child creation loop, once the node has been attached to the parent, we traverse into that node by passing its pointer, its polygon and detail area lists, and its bounding box into another recursion of the BuildTree method.

We have now covered all the code to build an oct-tree. It should be clear to you that having the class hierarchy and the housekeeping methods tucked away in CBaseTree and CBaseLeaf, makes creating new tree types simple and requires very little code.

14.19.3 The Oct-Tree Query Methods

The methods to query the oct-tree are almost identical to their quad-tree counterparts, so we will cover them only briefly. Each function works in exactly the same way with the exception that eight children will need to be tested and traversed into instead of four.

CollectLeavesAABB - COctTree

The CollectLeavesAABB method is actually identical to its quad-tree counterpart. It is passed an empty leaf list and a bounding box describing the query volume. The function simply wraps a call to the CollectAABBRecurse method for the root node.

```
bool COctTree::CollectLeavesAABB( LeafList & List,
                                  const D3DXVECTOR3 & Min,
                                  const D3DXVECTOR3 & Max )
{
    // Trigger Recursion
    return CollectAABBRecurse( m_pRootNode, List, Min, Max );
}
```

CollectAABBRecurse – COctTree

This recursive method steps through every leaf in the tree that is contained or partially contained in the passed query volume. For a leaf found inside the query volume, its pointer is added to the passed leaf list so it can be returned to the application. If it is determined at any node that its volume does not intersect the query volume we can refrain from further traversing that branch of the tree. Otherwise, we must traverse into any child nodes that do intersect the query volume. As soon as a leaf is found, its pointer is added to the passed leaf list. If at any point the query volume is found to completely contain the node volume, we know that all the children of that node must also be contained. In such a situation the bAutoCollect boolean is set to true and for every child node under the contained node, we no longer perform AABB tests and automatically step into each of its children until the leaf nodes are encountered.

```
bool COctTree::CollectAABBRecurse( COctTreeNode * pNode,
                                   LeafList & List,
                                   const D3DXVECTOR3 & Min,
                                   const D3DXVECTOR3 & Max,
                                   bool bAutoCollect /* = false */ )
{
    bool  bResult = false;
    ULONG i;

    // Validate parameters
    if ( !pNode ) return false;

    // Does the specified box intersect this node?
```

```
    if ( !bAutoCollect && !CCollision::AABBIntersectAABB( bAutoCollect,
                                                          Min,
                                                          Max,
                                                          pNode->BoundsMin,
                                                          pNode->BoundsMax ) )
                                                          return false;

    // Is there a leaf here, add it to the list
    if ( pNode->Leaf ) { List.push_back( pNode->Leaf ); return true; }

    // Traverse down to children
    for ( i = 0; i < 8; ++i )
    {
        if ( CollectAABBRecurse( pNode->Children[i],
                                 List,
                                 Min,
                                 Max,
                                 bAutoCollect ) ) bResult = true;

    } // Next Child

    // Return the 'was anything added' result.
    return bResult;
}
```

So the only difference between this method and its quad-tree counterpart is the fact that it loops through eight children instead of four. You should be able to see a pattern forming here that will allow you to create any tree type you desire by implementing only a few core functions.

CollectLeavesRay - COctTree

The CollectLeavesRay method is called by the application to query the oct-tree for any leaves that intersect the passed ray. This method is simply a wrapper around a call to the CollectRayRecurse method for the root node. It is unchanged from its quad-tree counterpart.

```
bool COctTree::CollectLeavesRay( LeafList & List,
                                 const D3DXVECTOR3 & RayOrigin,
                                 const D3DXVECTOR3 & Velocity )
{
    return CollectRayRecurse( m_pRootNode, List, RayOrigin, Velocity );
}
```

CollectRayRecurse - COctTree

This method is almost unchanged from its quad-tree counterpart with the exception that it steps into eight children at each non-terminal node instead of four.

```
bool COctTree::CollectRayRecurse( COctTreeNode * pNode,
                                  LeafList & List,
                                  const D3DXVECTOR3 & RayOrigin,
                                  const D3DXVECTOR3 & Velocity )
{
    bool  bResult = false;
    ULONG i;
    float t;

    // Validate parameters
    if ( !pNode ) return false;

    // Does the ray intersect this node?
    if ( !CCollision::RayIntersectAABB( RayOrigin,
                                        Velocity,
                                        pNode->BoundsMin,
                                        pNode->BoundsMax,
                                        t ) ) return false;


    // Is there a leaf here, add it to the list
    if ( pNode->Leaf ) { List.push_back( pNode->Leaf ); return true; }


    // Traverse down to children
    for ( i = 0; i < 8; ++i )
    {
        if ( CollectRayRecurse( pNode->Children[i],
                                List,
                                RayOrigin,
                                Velocity ) ) bResult = true;

    } // Next Child

    // Return the 'was anything added' result.
    return bResult;
}
```

And there we have it. We have implemented the query methods for the oct-tree with only a few changes to the code for the quad-tree.

14.19.4 The Oct-Tree Debug Draw Routine

As you have seen, most of the oct-tree functions were essentially the same as their quad-tree counterparts with the exception that eight children have to be processed at each node instead of four.

Because of this, we will not spend time showing the code to the COctTree::DebugDraw method even though it has been implemented in the source code. If you examine the code you will see that it is almost identical to its quad-tree counterpart with the exception that it steps into eight children.

This concludes our discussion and implementation of oct-trees for this lesson. We now have three tree types (quad-tree, YV quad-tree, oct-tree) to choose from for use as a spatial manager in our future applications. We have just one more tree type to create in this lesson: the kD-tree.

14.20 kD-Tree Implementation

The kD-tree is really the simplest tree to implement due to the fact that we only have a single plane used to partition space at each node. Implementing the kD-tree will once again involve deriving a class from CBaseTree and implementing the core functions to build and query the tree. The kD-tree class declaration and implementation can be found in the project source files CKDTree.h and CKDTree.cpp. As with all tree types, we will implement a node type specific to the tree type.

Although the kD-tree object's query function could be implemented in exactly the same way as the quad-tree and oct-tree versions (only with two children instead of one) we will use the CKDTree::CollectLeavesRay method to implement a traversal strategy closer to what we will see when we introduce the BSP tree. You will recall that while, technically speaking, a kD-tree is a BSP tree because it partitions space into two halfspaces at each node, the kD-tree carves the world into axis aligned bounding boxes that fit neatly into each leaf node. Therefore, with a kD-tree, just like an oct-tree and a quad-tree, the query methods that traverse the tree can locate the leaves that a ray intersects by performing ray/box intersection tests if desired. However, there is an alternative that we will explore that does not use box testing.

A BSP tree (see Chapters 16 and 17) is a binary tree that partitions space into two halfspaces at each node just like the kD-tree. However, the split planes of a BSP tree do not have to be axis aligned; in fact they are usually based on the polygon data being compiled. While this will all make a lot more sense later in the course (and beyond, when we really get into using BSP trees), just know for now that the leaves of a BSP tree will not all be box shaped. The leaves can be arbitrarily shaped convex hulls which cannot be neatly represented with an AABB. Therefore, we cannot actually traverse a BSP tree in quite the same way when we want to find the leaves. The CollectLeavesRay method of the kD-tree will be a more BSP-centric version that traverses the tree performing a series of ray/plane tests instead of ray/box tests. This will give us some early insight into the BSP traversals we will be performing in later chapters. However, it is worth noting that in the case of the kD-tree, where the leaves are all box shaped, you could replace our CollectLeavesRay method with a version that is almost identical to the ones we have seen for other tree types.

14.20.1 CKDTreeNode – The Source Code

Each node in our KD-tree will be an object of type CKDTreeNode which is implemented in the project source files CKDTree.h and CKDTree.cpp. Just like the other node types we have seen it stores a bounding box and a pointer to a CBaseLeaf object which will be used only by terminal nodes. Because a kD-tree only has two children, we will not store the child pointers in an array but will instead simply have two pointers called Front and Back. The names of these children obviously describe its position relative to the parent node's single split plane. Like all other node types, we implement the recursive SetVisible method so that the tree can inform a node to start a recursive process that will flag all child leaves of the node as either visible or invisible. Finally, notice in the following class declaration that unlike the quad-tree and oct-tree implementations, where the node split planes were created temporarily and then discarded, in the case of the kD-tree, we have decided to store the single split plane used to partition the node's space during the build process.

```
class CKDTreeNode
{
public:

    // Constructors & Destructors for This Class.
     CKDTreeNode( );
    ~CKDTreeNode( );

    // Public Functions for This Class
    void SetVisible( bool bVisible );

    // Public Variables for This Class
    D3DXPLANE       Plane;              // Splitting plane for this node
    CKDTreeNode *   Front;              // Node in front of the plane
    CKDTreeNode *   Back;               // Node behind the plane
    CBaseLeaf   *   Leaf;               // If this is a leaf, store here.
    D3DXVECTOR3     BoundsMin;          // Minimum bounding box extents
    D3DXVECTOR3     BoundsMax;          // Maximum bounding box extents
    signed char     LastFrustumPlane;   // The 'last plane' index.
};
```

The reason we have decided to store the node plane in the kD-tree case is because this is more in keeping with the traditional BSP approach. In a BSP tree, each node stores a single split plane and pointers to two children that represent the halfspaces of that plane. By storing the plane in the kD-tree node, we enable our kD-tree to have the ability to work more like a traditional BSP tree if we so desire (again, in many respects they are basically the same type of tree, so this works quite well).

For example, in the CollectLeavesAABB method we will implement it using the same style as the quad-tree and oct-tree. The tree will be traversed with a query volume that will be tested for intersection against the bounding box of each child. However, in the CollectLeavesRay method we will implement the method just as we would for a BSP tree. We will assume that we have no AABB stored at each node; only a split plane. This method will send the ray down the tree and at each node test to see if the ray is spanning the node. If it is, the ray is split in two and each half of the ray is sent into the respective child for the halfspace the ray segment is contained within. Eventually, all of our split ray fragments will pop out in leaf nodes. At that point, the leaves are added to the collection list (more on this later). In order for

such a method to be implemented we need to have a tree where each node represents a split plane, just like a BSP tree. By storing the split plane in the kD-tree node and implementing our ray query method as a BSP method, we draw attention to the similarities between the kD-tree and the BSP tree. This should prove to be helpful when we move on to examine BSP trees.

SetVisible - CKDTreeNode

The SetVisible method of the kD-tree's node object is the simplest we have seen so far. If the current node being visited points to a leaf object, then this is a terminal node and we set the visibility status of the attached leaf and return. Otherwise, we are in a normal kD-tree node and we must traverse into both children.

```
void CKDTreeNode::SetVisible( bool bVisible )
{
    // Set leaf property
    if ( Leaf  ) { Leaf->SetVisible( bVisible ); return; }

    // Recurse down front / back if applicable
    if ( Front ) Front->SetVisible( bVisible );
    if ( Back  ) Back->SetVisible( bVisible );
}
```

This function is simpler than the oct-tree and quad-tree case because we have only two children to process and no need to loop.

14.20.2 The CKDTree Source Code

The CKDTree class looks almost identical to the quad-tree and oct-tree class declarations due to the fact that it is derived from CBaseTree and has to implement all the same building and querying methods.

```
class CKDTree : public CBaseTree
{
public:

    // Constructors & Destructors for This Class.
    virtual ~CKDTree();
             CKDTree( LPDIRECT3DDEVICE9 pDevice,
                      bool  bHardwareTnL,
                      float fMinLeafSize = 300.0f,
                      ULONG nMinPolyCount = 600,
                      ULONG nMinAreaCount = 0 );

    // Public Virtual Functions for This Class (from base).
    virtual bool Build                 ( bool bAllowSplits = true );

    virtual void ProcessVisibility     ( CCamera & Camera );
```

```
    virtual bool CollectLeavesAABB      ( LeafList & List,
                                          const D3DXVECTOR3 & Min,
                                          const D3DXVECTOR3 & Max );

    virtual bool CollectLeavesRay       ( LeafList & List,
                                          const D3DXVECTOR3 & RayOrigin,
                                          const D3DXVECTOR3 & Velocity );

    virtual void DebugDraw              ( CCamera & Camera );

    virtual bool GetSceneBounds         ( D3DXVECTOR3 & Min, D3DXVECTOR3 & Max );

protected:

    // Protected virtual Functions for This Class
    virtual bool BuildTree              ( CKDTreeNode * pNode,
                                          PolygonList PolyList,
                                          DetailAreaList AreaList,
                                          const D3DXVECTOR3 & BoundsMin,
                                          const D3DXVECTOR3 & BoundsMax,
                                          ULONG PlaneType = 0 );


    // Protected Functions for This Class
    void         UpdateTreeVisibility   ( CKDTreeNode * pNode,
                                          CCamera & Camera,
                                          UCHAR FrustumBits = 0x0 );

    bool         DebugDrawRecurse       ( CKDTreeNode * pNode,
                                          CCamera & Camera,
                                          bool bRenderInLeaf );

    bool         CollectAABBRecurse     ( CKDTreeNode * pNode,
                                          LeafList & List,
                                          const D3DXVECTOR3 & Min,
                                          const D3DXVECTOR3 & Max,
                                          bool bAutoCollect = false );

    bool         CollectRayRecurse      ( CKDTreeNode * pNode,
                                          LeafList & List,
                                          const D3DXVECTOR3 & RayOrigin,
                                          const D3DXVECTOR3 & Velocity );


    // Protected Variables for This Class

    CKDTreeNode * m_pRootNode;          // The root node of the tree
    bool          m_bAllowSplits;       // Is splitting allowed?
    float         m_fMinLeafSize;       // Min leaf size stop code
    ULONG         m_nMinPolyCount;      // Min polygon count stop code
    ULONG         m_nMinAreaCount;      // Min detail area count stop code

};
```

As you can see, the only difference in the member variable list is that it now stores a root node pointer of type CKDTreeNode.

Build - CKDTree

We now know that the Build function of our derived classes is responsible for allocating the root node and calculating its bounding box. It then begins the recursive building process, starting at the root, with a call to the BuildTree method.

The Build method of the kD-tree is almost identical to the others we have seen, so we will cover it only briefly here.

The function first allocates a new CKDTreeNode object and assigns the tree's root node pointer to point at it. We then store that passed boolean parameter so that we know during the building process whether or not we are creating a clipped tree.

```
bool CKDTree::Build( bool bAllowSplits /* = true */ )
{
    PolygonList::iterator       PolyIterator = m_Polygons.begin();
    DetailAreaList::iterator    AreaIterator = m_DetailAreas.begin();
    PolygonList                 PolyList;
    DetailAreaList              AreaList;
    unsigned long               i;

    // Reset our tree info values.
    D3DXVECTOR3 vecBoundsMin( FLT_MAX, FLT_MAX, FLT_MAX );
    D3DXVECTOR3 vecBoundsMax( -FLT_MAX, -FLT_MAX, -FLT_MAX );

    // Allocate a new root node
    m_pRootNode = new CKDTreeNode;
    if ( !m_pRootNode ) return false;

    // Store the allow splits value for later retrieval.
    m_bAllowSplits = bAllowSplits;
```

We now need to calculate the bounding box for the root node so we loop through every vertex of every polygon registered with the tree and expand the box to fit all vertices. As we process each polygon, its pointer is also copied into the local polygon list (PolyList). After we have resized the box for every polygon and made a copy of the original tree's polygon list, we test to see if a clipped tree is being built. If so, we empty the tree's polygon list since it will be populated with polygon fragments during clipping.

```
    // Loop through all of the initial polygons
    for ( ; PolyIterator != m_Polygons.end(); ++PolyIterator )
    {
        // Retrieve the polygon
        CPolygon * pPoly = *PolyIterator;
        if ( !pPoly ) continue;

        // Calculate total scene bounding box.
        for ( i = 0; i < pPoly->m_nVertexCount; ++i )
        {
            // Store info
            CVertex * pVertex = &pPoly->m_pVertex[i];
            if ( pVertex->x < vecBoundsMin.x ) vecBoundsMin.x = pVertex->x;
```

```
            if ( pVertex->y < vecBoundsMin.y ) vecBoundsMin.y = pVertex->y;
            if ( pVertex->z < vecBoundsMin.z ) vecBoundsMin.z = pVertex->z;
            if ( pVertex->x > vecBoundsMax.x ) vecBoundsMax.x = pVertex->x;
            if ( pVertex->y > vecBoundsMax.y ) vecBoundsMax.y = pVertex->y;
            if ( pVertex->z > vecBoundsMax.z ) vecBoundsMax.z = pVertex->z;
        }

        // Store this polygon in the top polygon list
        PolyList.push_back( pPoly );

    } // Next Polygon

    // Clear the initial polygon list if we are going to split the polygons
    // as this will eventually become storage for whatever gets built
    if ( bAllowSplits ) m_Polygons.clear();
```

The bounding box is large enough to contain all registered polygons, but we also need to factor in any registered detail areas.

```
    // Loop through all of the detail areas
    for ( ; AreaIterator != m_DetailAreas.end(); ++AreaIterator )
    {
        // Retrieve the detail area
        TreeDetailArea * pDetailArea = *AreaIterator;
        if ( !pDetailArea ) continue;

        // Calculate total scene bounding box.
        D3DXVECTOR3 & Min = pDetailArea->BoundsMin;
        D3DXVECTOR3 & Max = pDetailArea->BoundsMax;
        if ( Min.x < vecBoundsMin.x ) vecBoundsMin.x = Min.x;
        if ( Min.y < vecBoundsMin.y ) vecBoundsMin.y = Min.y;
        if ( Min.z < vecBoundsMin.z ) vecBoundsMin.z = Min.z;
        if ( Max.x > vecBoundsMax.x ) vecBoundsMax.x = Max.x;
        if ( Max.y > vecBoundsMax.y ) vecBoundsMax.y = Max.y;
        if ( Max.z > vecBoundsMax.z ) vecBoundsMax.z = Max.z;

        // Store this in the top detail area list
        AreaList.push_back( pDetailArea );

    } // Next Polygon
```

At this point we call the BuildTree method to start the tree building process from the root node. The function is passed the root node pointer and a list of polygons and detail areas that are contained in the root node's bounding box. The final two parameters represent the AABB that bounds all data contained in the root node.

```
    // Build the tree itself
    if ( !BuildTree( m_pRootNode, PolyList, AreaList, vecBoundsMin,vecBoundsMax))
         return false;

    // Allow our base class to finish any remaining processing
    return CBaseTree::PostBuild( );
}
```

When the BuildTree method returns and the entire tree has been compiled, we remember to call the CBaseTree::PostBuild method to give the base class a chance to initialize the rendering system and calculate the AABBs for each polygon in the tree.

BuildTree - CKDTree

This function is a little different from the others we have seen so far. Not only does it split the node using a single plane and compile two lists of polygon and detail area data (one for each child), the plane used to split the node is actually switched between iterations. That is, at the first node we split the node using a plane whose normal faces down the world Z axis. When we step into each of the children, we flip the plane so that it is now aligned with the world X axis. For each of its children the plane is flipped again so that its normal is facing along the world Y axis. Therefore, although at each node we are partitioning space into only two children, the scene is carved up in a very similar way to an oct-tree. As we step through the levels of the kD-tree, the plane is continuously alternated between the three world axis aligned planes, wrapping around to the first plane used after every three levels of depth.

> **Note:** You can also use a kD-tree to partition 2D space. Simply transition between the two planes you are interested in rather than three planes.

The Build function is actually a bit simpler than the others we have seen, although at first glance it may not appear so because it is a little larger. The only reason why this is the case is because we are not generating the child nodes in a loop since there are only two children.

The kD-tree BuildTree method accepts an additional plane flag parameter which defaults to zero the first time it is called for the root node. Every time this method calls itself recursively, this parameter will be set in the range [0, 2] describing which of the three world aligned planes should be used to partition the node. The first time it is called from the Build method no value is passed, which means it is set to zero for the root node. This tells the function that the root node should be divided using the first of the three split planes; the split plane whose normal is pointing down the world z axis. Every time this method calls itself to traverse into children, it will increment this value (wrapping around to zero once it is greater than 2) such that the next world aligned plane will be used for its children. For example, the root node will use plane 0 which is the plane aligned with the world Z axis. Its children will be passed a plane value of 1 describing that they should be partitioned using the plane aligned with the world X axis, and so on.

The first section of the code stores the passed bounding box in the passed node and calculates the size of the node's bounding box. It then uses this box and the number of polygons and detail areas passed into the node to determine if this node should become a leaf.

```
bool CKDTree::BuildTree(      CKDTreeNode * pNode,
                              PolygonList PolyList,
                              DetailAreaList AreaList,
                              const D3DXVECTOR3 & BoundsMin,
                              const D3DXVECTOR3 & BoundsMax,
                              ULONG PlaneType /* = 0 */ )
{
```

```
    D3DXVECTOR3                 Normal;
    PolygonList::iterator       PolyIterator;
    DetailAreaList::iterator    AreaIterator;
    CPolygon                  * CurrentPoly, * FrontSplit, * BackSplit;
    PolygonList                 FrontList, BackList;
    DetailAreaList              FrontAreaList, BackAreaList;
    bool                        bStopCode;

    // Store the bounding box properties in the node
    pNode->BoundsMin = BoundsMin;
    pNode->BoundsMax = BoundsMax;

    // Calculate 'Stop' code
    D3DXVECTOR3 vecLeafSize = BoundsMax - BoundsMin;
    bStopCode  = (AreaList.size() == 0 && PolyList.size() == 0) ||
                 (AreaList.size() <= m_nMinAreaCount &&
                  PolyList.size() <= m_nMinPolyCount) ||
                 D3DXVec3Length( &vecLeafSize ) <= m_fMinLeafSize;
```

As with all other tree types, if the stop code boolean gets set to true, we know we have reached a terminal node. The code for creating the leaf is exactly the same as all other tree types. We create a new CBaseLeaf object and then add every polygon and detail area that made it into this node to the leaf. If a clipped tree is being built, then we also re-add the polygon data that made it into this node to the tree's main polygon list. We then attach the leaf to the node, store the node's bounding box in the leaf, and add the leaf to the tree's leaf list before returning.

```
    // If we reached our stop limit, build a leaf here
    if ( bStopCode )
    {
        // Build a leaf
        CBaseLeaf * pLeaf = new CBaseLeaf( this );
        if ( !pLeaf ) return false;

        // Store the polygons
        for (     PolyIterator = PolyList.begin();
                  PolyIterator != PolyList.end(); ++PolyIterator )
        {
            // Retrieve poly
            CurrentPoly = *PolyIterator;
            if ( !CurrentPoly ) continue;

            // Add to full tree polygon list ONLY if splitting was allowed
            if ( m_bAllowSplits ) AddPolygon( CurrentPoly );

            // Also add a reference to the leaf's list
            pLeaf->AddPolygon( CurrentPoly );

        } // Next Polygon

        // Store the area lists
        for ( AreaIterator = AreaList.begin();
              AreaIterator != AreaList.end(); ++AreaIterator )
        {
            // Retrieve detail area item
            TreeDetailArea * pDetailArea = *AreaIterator;
```

```
            if ( !pDetailArea ) continue;

            // Add a reference to the leaf's list
            pLeaf->AddDetailArea( pDetailArea );

        } // Next Polygon

        // Store pointer to leaf in the node and the bounds
        pNode->Leaf = pLeaf;

        // Store the bounding box in the leaf
        pLeaf->SetBoundingBox( pNode->BoundsMin, pNode->BoundsMax );

        // Store the leaf in the leaf list
        AddLeaf( pLeaf );

        // We have reached a leaf
        return true;

    } // End if reached stop code
```

If we reach this point in the function, we know we are not at a leaf node and we will have to further partition this node into two halfspaces using the split plane. Which of the three axis aligned split planes we use to create the split plane is determined by the PlaneType parameter passed in. Notice in the next code block that this time we actually store the split plane in the node's Plane member.

```
    // Otherwise we must continue to refine. Choose a plane.
    if ( PlaneType == 0 )
        Normal = D3DXVECTOR3( 0.0f, 0.0f, 1.0f );
    else if ( PlaneType == 1 )
        Normal = D3DXVECTOR3( 1.0f, 0.0f, 0.0f );
    else if ( PlaneType == 2 )
        Normal = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );

    // Generate the actual plane
    D3DXPlaneFromPointNormal( &pNode->Plane,
                              &((BoundsMin + BoundsMax) / 2.0f),
                              &Normal );
```

Now that we have determined which split plane to use at this node, we will loop through every polygon in the node's polygon list and classify it against the plane. If it is in front of the plane we will add it to the front polygon list (the list that will be passed to the front child) and if it is behind the plane we will add it to the back list. In the case where the polygon is on the plane, it does not really matter which child we assign it to (we will add it to the front list).

```
    // Classify all polygons
    for (   PolyIterator = PolyList.begin();
            PolyIterator != PolyList.end(); ++PolyIterator )
    {
        // Store current poly
        CurrentPoly = *PolyIterator;

        // Classify the poly against the plane
```

```
        CCollision::CLASSIFYTYPE Location = CCollision::PolyClassifyPlane
                                                    (CurrentPoly->m_pVertex,
                                                     CurrentPoly->m_nVertexCount,
                                                     sizeof(CVertex),
                                                     (D3DXVECTOR3&)pNode->Plane,
                                                     pNode->Plane.d );

        // Decide what to do
        switch ( Location )
        {
            case CCollision::CLASSIFY_BEHIND:

                // Add straight to the back list
                BackList.push_back( CurrentPoly );
                break;

            case CCollision::CLASSIFY_ONPLANE:
            case CCollision::CLASSIFY_INFRONT:

                // Add straight to the front list
                FrontList.push_back( CurrentPoly );
                break;
```

If the polygon is spanning the plane and we are building a clipped tree, we will split the polygon into two new child polygons and delete the original polygon. We will then add the front split polygon to the front list and the back split polygon to the back list. If the polygon is spanning but we are not building a clipped tree, we will simply assign the polygon to the front and back lists to that it gets assigned to both child nodes in which it is partially contained.

```
            case CCollision::CLASSIFY_SPANNING:

                // Is splitting allowed?
                if ( m_bAllowSplits )
                {
                    // Split the current poly against the plane and delete it
                    CurrentPoly->Split( pNode->Plane, &FrontSplit, &BackSplit );
                    delete CurrentPoly;

                    // Add the fragments (if any survived) to the appropriate lists
                    if ( BackSplit  ) BackList.push_back( BackSplit );
                    if ( FrontSplit ) FrontList.push_back( FrontSplit );
                }
                else
                {
                    // Add to both lists!
                    FrontList.push_back( CurrentPoly );
                    BackList.push_back( CurrentPoly );
                }
                break;
        }
    }
```

With the polygon lists compiled for both children, we will classify the detail areas against the plane and add them to either the front or back list (or both). If the detail area is behind the plane it is added to the

back list of detail objects and if it is in front of the plane it is added to the front detail list. If the detail area's AABB is spanning the plane, it is added to the detail area list of both children.

```
    // Classify all detail areas
    for ( AreaIterator = AreaList.begin();
          AreaIterator != AreaList.end();
        ++AreaIterator )
    {
        // Store current area
        TreeDetailArea * pDetailArea = *AreaIterator;

        // Classify the area against the plane
        CCollision::CLASSIFYTYPE Location = CCollision::AABBClassifyPlane
                                                       (pDetailArea->BoundsMin,
                                                        pDetailArea->BoundsMax,
                                                        (D3DXVECTOR3&)pNode->Plane,
                                                        pNode->Plane.d );

        // Decide what to do
        switch ( Location )
        {
            case CCollision::CLASSIFY_BEHIND:

                // Add straight to the back list
                BackAreaList.push_back( pDetailArea );
                break;

            case CCollision::CLASSIFY_INFRONT:

                // Add straight to the front list
                FrontAreaList.push_back( pDetailArea );
                break;

            case CCollision::CLASSIFY_SPANNING:

                // Add to both the front and back lists
                BackAreaList.push_back( pDetailArea );
                FrontAreaList.push_back( pDetailArea );
                break;

        } // End Switch

    } // Next Detail Area
```

At this point we have the polygon lists and the detail area lists compiled for each child node that we are about to create. Unlike the same method from previous tree types, we will not perform child creation in a loop (we will unroll that loop). This next section of code builds the front child first. It calculates its bounding volume based on the split plane being used by the parent node and then allocates a new CKDTreeNode object which is assigned to the front child pointer of the parent node. We then recur into that node by calling the BuildTree function again for the front child.

```
    D3DXVECTOR3 NewBoundsMin, NewBoundsMax, MidPoint;

    // Calculate box midpoint
```

```
    MidPoint = (BoundsMin + BoundsMax) / 2.0f;

    // Calculate child bounding box values
    NewBoundsMax = BoundsMax;

    if ( PlaneType == 0 ) // XY Plane
        NewBoundsMin = D3DXVECTOR3( BoundsMin.x, BoundsMin.y, MidPoint.z);
    else if ( PlaneType == 1 ) // XZ Plane
        NewBoundsMin = D3DXVECTOR3( MidPoint.x, BoundsMin.y, BoundsMin.z);
    else if ( PlaneType == 2 ) // YZ Plane
        NewBoundsMin = D3DXVECTOR3( BoundsMin.x, MidPoint.y, BoundsMin.z);

    // Allocate child node
    pNode->Front = new CKDTreeNode;
    if ( !pNode->Front ) return false;

    // Recurse into this new node
    BuildTree( pNode->Front,
               FrontList,
               FrontAreaList,
               NewBoundsMin,
               NewBoundsMax,
              (PlaneType + 1) % 3 );

    // Clean up
    FrontList.clear();
    FrontAreaList.clear();
```

As you can see, the above code is calculating the bounding box for the child that is in the front half space of the node. This basically means the child node's bounding volume will be one half of the parent node's volume. If the parent node split its space using the XY plane, the minimum extent of the bounding box will be minx, miny and minz as this describes the minimum extents of a box that starts halfway along the parent node's Z axis (see Figure 14.81). In the case of the YZ plane, the front child's bounding box will be the right half of the parent node, where the minimum X extent will be in the center of the box. In each case, because of the way the normals of the planes are facing, the maximum extents of the front child will always be the maximum extents of the parent node (see Figures 14.81 through 14.83). Figure 14.83 similarly shows how the minimum and maximum extents of the front child's AABB would be calculated using the parent node's bounding box if the parent node partitioned space using the XZ plane.

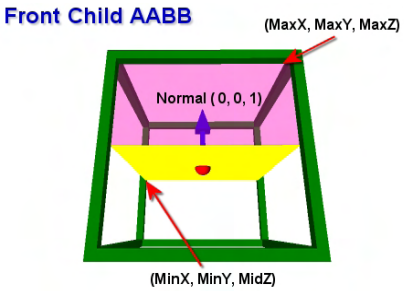


Figure 14.81
Front child of the XY plane.

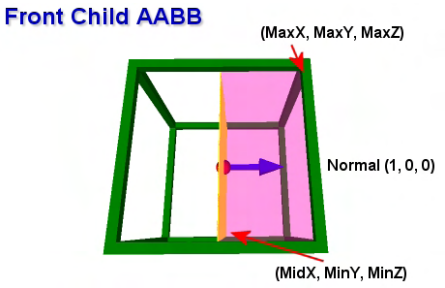


Figure 14.82
Front child of the YZ plane.

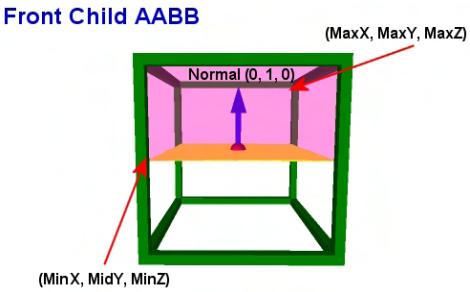


Figure 14.83
Front Child of the XZ Plane.

After we determine which plane was used in the above code we calculate the bounding box for the front child and allocate the new node. The node is then attached to the parent node's front pointer. The BuildTree function then calls itself to step into the front child node and continue the build process down that branch of the tree. Notice that when we call the BuildTree function, we increment the current value of the PlaneType parameter so that the child node will use the next plane in the list of three to partition its space. Also note that we mod this with the number 3 so that as soon as we increment PlaneCount beyond a value of 2, it wraps around to 0 again to start the plane selection process from the beginning. That is, in the child node, the XY plane will be chosen as the split plane.

When the BuildTree method returns above, the front child and all its child nodes and leaves will have been created. Now it is time to build the back child (i.e., the child that is positioned in the back halfspace of the parent node's split plane).

The first thing we do is calculate the bounding box extents of the back child based on the plane that was used as a splitter for the parent node. Because this child is on the back of the parent node's split plane, the bounding box extents of the child will range from min to midpoint on the axis aligned with the plane. The minimum extents of the back child will always be equal to the minimum extents of the parent node's volume for all components, regardless of the split plane being used.

```
    // Calculate child bounding box values
    NewBoundsMin = BoundsMin;

    if ( PlaneType == 0 ) // XY Plane
        NewBoundsMax = D3DXVECTOR3( BoundsMax.x, BoundsMax.y, MidPoint.z);
    else if ( PlaneType == 1 ) // XZ Plane
        NewBoundsMax = D3DXVECTOR3( MidPoint.x, BoundsMax.y, BoundsMax.z);
    else if ( PlaneType == 2 ) // YZ Plane
        NewBoundsMax = D3DXVECTOR3( BoundsMax.x, MidPoint.y, BoundsMax.z);
```

If you are having trouble picturing what we are doing then take a look at Figures 14.84 through 14.86. The figures depict how the bounding box of the back child is calculated using the components of the parent node's volume based on chosen split plane.

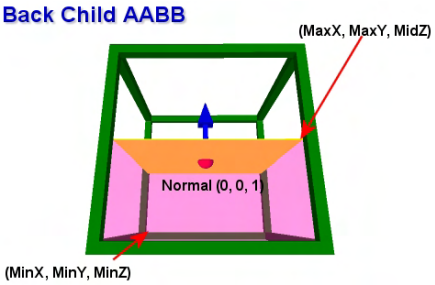


Figure 14.84
Back child of the XY plane.

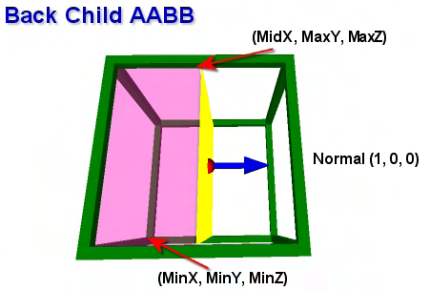


Figure 14.85
Back child of the YZ plane.

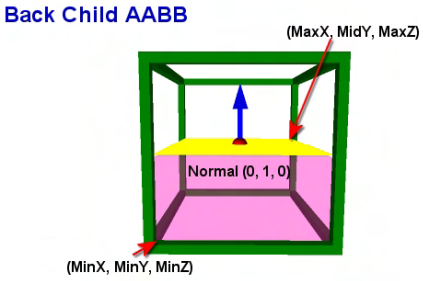


Figure 14.86
Back child of the XZ plane

With the bounding box calculated for the back child we allocate a new CKDTreeNode and attach it to the parent node's Back child pointer. We then call the BuildTree method and recur into the back child passing in the polygon and detail area lists for that node along with its bounding volume. Once again, we increment the PlaneType value that was passed into this instance of the function so that in the next level of the tree we use the next split plane. The modulus is performed so that we wrap this value around to 0 should it exceed 2.

```
    // Allocate child node
    pNode->Back = new CKDTreeNode;
    if ( !pNode->Back ) return false;

    // Recurse into this new node
    BuildTree( pNode->Back,
               BackList,
               BackAreaList,
               NewBoundsMin,
               NewBoundsMax,
               (PlaneType + 1) % 3 );

    // Clean up
    BackList.clear();
    BackAreaList.clear();

    // Success!
    return true;
}
```

This completes the build code for the kD-tree, the last tree type we will have to compile in this lesson. We have now learned how to build quad-trees, oct-trees and kD-trees. These trees should turn out to be very useful throughout your game making career. Indeed, we will be using such trees in virtually all of our lab projects moving forward.

14.20.3 The kD-Tree Query Methods

Querying the kD-tree can be done in a nearly identical manner to quad-trees and oct-trees, with the exception that only two children will need to be tested and traversed during leaf collection. However, although CollectLeavesAABB (and its recursive helper function) will be implemented in an almost

identical manner to the previous versions we have covered for other trees, we have chosen to implement the CollectLeavesRay method (and its recursive helper function) to test using the node split plane instead of the bounding box. This need not be done this way because the box could be used instead, but it does give us an opportunity to introduce a strategy that will be used by its BSP counterpart (BSP trees are generally traversed using the node planes). This will help prepare you for the querying operations we will perform on BSP trees, where the leaves do not fit neatly into axis aligned bounding boxes as we have seen with the trees in this lesson.

CollectLeavesAABB - CKDTree

This method is identical to those implemented in all other tree classes. It simply wraps the call to the recursive CollectAABBRecurse method for the root node. It is called by the application to start the traversal at the root node, finding and collecting all leaves that intersect the query volume.

```
bool CKDTree::CollectLeavesAABB( LeafList & List,
                                 const D3DXVECTOR3 & Min,
                                 const D3DXVECTOR3 & Max )
{
    // Trigger Recursion
    return CollectAABBRecurse( m_pRootNode, List, Min, Max );
}
```

CollectAABBRecurse – CKDTree

This method is called by the previous method to visit the nodes of the tree and step into any children that are inside (or partially inside) the query volume. It is implemented in an identical manner to the other versions of this method for the other tree types with the exception that when a node is not a leaf, we step into only two children.

```
bool CKDTree::CollectAABBRecurse( CKDTreeNode * pNode,
                                  LeafList & List,
                                  const D3DXVECTOR3 & Min,
                                  const D3DXVECTOR3 & Max,
                                  bool bAutoCollect /* = false */ )
{
    bool bResult = false;

    // Validate parameters
    if ( !pNode ) return false;

    // Does the specified box intersect this node?
    if ( !bAutoCollect && !CCollision::AABBIntersectAABB( bAutoCollect,
                                                          Min,
                                                          Max,
                                                          pNode->BoundsMin,
                                                          pNode->BoundsMax ))
           return false;
```

```
    // Is there a leaf here, add it to the list
    if ( pNode->Leaf ) { List.push_back( pNode->Leaf ); return true; }

    // Traverse down to children
    if ( CollectAABBRecurse( pNode->Front, List, Min, Max, bAutoCollect ))
         bResult = true;

    if ( CollectAABBRecurse( pNode->Back , List, Min, Max, bAutoCollect ))
         bResult = true;

    // Return the 'was anything added' result.
    return bResult;
}
```

Once again we use the bAutoCollect boolean to initiate immediate collection of children without further AABB testing.

CollectLeavesRay - CKDTree

The CollectLeavesRay method is also identical to previous versions. It wraps the call to the CollectRayRecurse method for the root node. The function is passed an empty leaf list and a ray origin and delta vector. On function return the passed leaf list will contain a list of all the leaves intersected by the ray.

```
bool CKDTree::CollectLeavesRay( LeafList & List,
                                const D3DXVECTOR3 & RayOrigin,
                                const D3DXVECTOR3 & Velocity )
{
    return CollectRayRecurse( m_pRootNode, List, RayOrigin, Velocity );
}
```

CollectRayRecurse - CKDTree

Unlike prior versions, we will implement this query using the split plane stored at each node instead of the bounding box. These are the types of queries we will be implementing when we start relying on BSP trees in the later lessons of this course. It also tends to be more efficient to perform a single ray plane test rather than a ray box test.

The basic process is to send a ray into a node and perform a ray/plane intersection with the split plane stored there. If the ray is contained completely in the front space of the plane, the function calls itself recursively passing the ray into the front child. If the ray is contained in the back space of the node we send the ray into the back child. However, if the ray is spanning the node we do *not* simply pass it down both the front and back children since this would certainly return leaves that the ray is not intersecting. Instead we will split the ray on the plane and send each ray fragment into its respective child node. Of course, the ray may get split into many fragments during its journey through the tree, but eventually

these ray fragments will pop out in leaf nodes for all leaves that were intersected by the original ray. Figure 14.87 depicts the basic ray splitting concept.

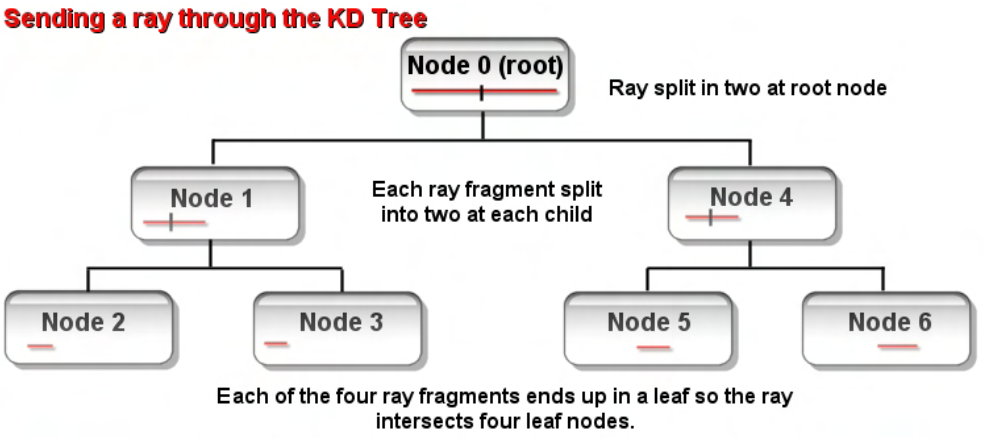


Figure 14.87

In this simple example, the kD-tree consists of seven nodes in total, with four of them being leaf nodes. The ray is assumed to intersect all leaf nodes in this example. Let us step through what is happening.

At the root node, the ray is classified against the plane and is found to be spanning that plane. The ray is split into two segments by this split plane creating two new rays, one in the plane's back space and the other in the front space. The ray fragment in the root node's front space is passed down into the front node (node 1). The ray fragment in node 1 is also found to be spanning its plane, so is further split into two new fragments. One fragment is passed to its front child (node 2) and the other is passed into its back child (node 3). When we reach node 2 and node 3, we find that the ray fragment has been passed into leaf nodes, so the original ray must intersect these leaves. The leaves for both nodes would thus be added to the leaf collection list. At this point the function would return and the recursive process will unroll back up the root node's traversal into its front child. Now the root node has to pass the other ray fragment (the back space fragment) into its back child (node 5). The ray fragment passed into node 5 is also spanning node 5's plane, so it is further split into two ray fragments which end up being passed into leaf nodes 6 and 7 for the front and back fragment, respectively. These are leaves are also added to the leaf collection list.

The above example had the ray was spanning the plane at every node to get across the idea of the recursive splitting process. However, if at any node the ray is found to be in *either* the front or back space, the ray is not split; it is simply passed into the respective child unaltered. In the end, we are essentially just sending the ray down the tree and collecting any leaves that contain any portion/fragment of the ray.

Clipping a ray that spans a plane into two fragments is very easy to do using the CCollision::RayIntersectPlane method we added to our collision library in the previous chapter. This function returns a *t* value of intersection that can be multiplied with the ray delta vector and added to the origin to calculate the actual point of intersection. Once we have the ray start and end points and the ray intersection point, it is easy to see which points define each ray fragment (see Figure 14.88).

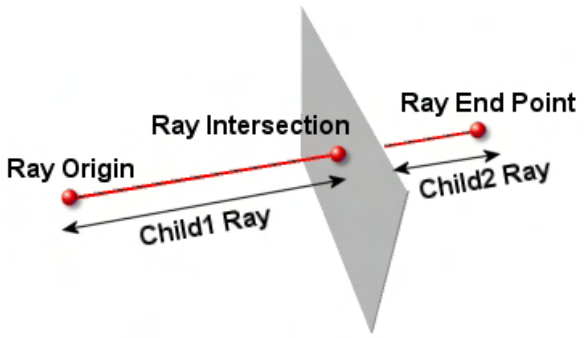


Figure 14.88

In this example we see a plane that intersects a ray. Once the intersection point with the plane is calculated, the ray in the left halfspace in this image is formed by the ray origin and the intersection point, and the ray fragment in the right halfspace is formed by the ray intersection point and the ray end point. That is, the ray intersection point forms the end of one fragment and the beginning of another. These two ray fragments would then be passed into the child nodes.

While splitting the ray and sending the fragments down the tree is pretty simple to implement, what might not be so obvious is why we need to split the ray at all. One might imagine that if the ray spans the plane we could simply send it in its entirety into both children. However, this will actually return incorrect leaves as shown in Figure 14.89.

Figure 14.89 shows a very simple kD-tree consisting of four leaf nodes and a ray that intersects three of those four leaves. In this example, we can assume that the split plane stored at the root node is labeled 'A'. During the building process it was determined that the child node attached to the front of the root (plane A) no longer needed to be subdivided, so the node was made a leaf. This leaf is labeled Leaf 1. When the building process processed the back child of the root (the space down the back of plane A) it was decided that this space should be further subdivided and as such a non-terminal node was created whose split plane is labeled 'B'. Down the front of B it was decided that we no longer needed to partition space, so Leaf 2 was created there. Down the back of B, another non-terminal node was created with the split plane labeled 'C', which further partitioned the top right quadrant of the tree into Leaf 3 and Leaf 4.

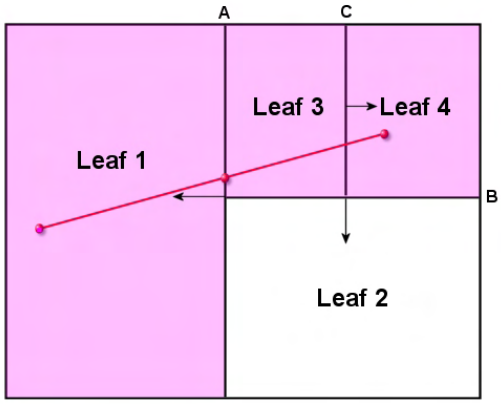


Figure 14.89

We can clearly see that the ray is contained in only three of the four leaf nodes (1, 3, and 4). However, let us see what would happen if we did not split the ray in the spanning case and just sent it through the tree unaltered. What we will learn is that Leaf 2 will be added to the collection list even though the ray does not intersect it. If Leaf 2 contains tens of thousands of polygons, that is a lot of polygons we would need to send to a narrow phase unnecessarily. Let us see why this is the case.

First we send the ray into the root node where it is classified against plane A. We find that it is spanning plane A (remember, we are not splitting in this example to demonstrate the flaw in that approach) so we send the ray into the front and back child of A. Down the front of A the ray ends up in Leaf 1 and it gets added to the list. So far, so good. Down the back of A we find we are at node B and once again classify the ray against plane B. Remembering that planes are infinite, we can see that the ray is actually spanning plane B (imagine B extending infinitely to the left and right). Because it is spanning B, we need to pass it into both children of B. Unfortunately, the front child of B is Leaf 2. Thus, we have just

allowed the ray to fall into Leaf 2 even though it does not intersect this leaf. We need go no further; already we can see that our approach is flawed.

Let us now see what happens if we split the ray at every spanning plane.

Looking again at Figure 14.89 we can see that when the ray is first passed into the root node, it is found to be spanning node plane A. We calculate the intersection point with the plane and create two child rays. The first ray is in the plane's front space and has the original ray's origin as its origin and the intersection point as its end point. The second ray is the ray contained in the plane's back space and has the intersection point as its origin and the original ray end point as its end point. We can see this relationship in Figure 14.90.

Figure 14.90 shows the intersection point between the ray and plane A. We create two new rays at this node. The ray fragment in the plane's front half space is fed into Leaf 1 and Leaf 1 is added to the list. The ray in the plane's back space is created from the ray intersection point with plane A and the original ray end point at the top right of the ray. When we feed this ray fragment into the back space of plane A, we find ourselves at the non-terminal node containing plane B. The important point here is that when we test the ray fed into node B, whose origin is the intersection point between the original ray and plane A, we see that the ray is completely behind plane B and therefore, we never traverse into the front space of B. Therefore, the ray never reaches Leaf 2 and Leaf 2 is never added to the collection list.

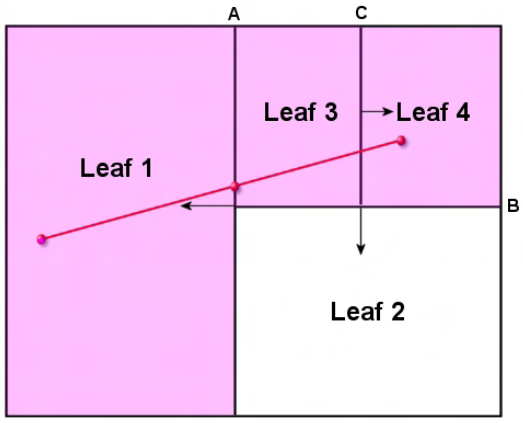


Figure 14.90

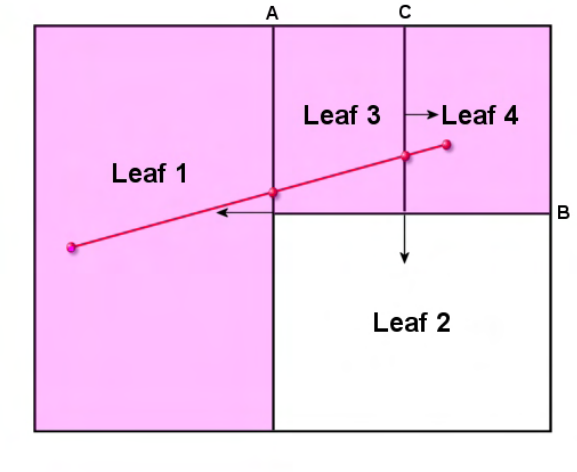


Figure 14.91

In Figure 14.90 the ray fragment was found to be completely behind plane B so it is passed into its child where it enters the non-terminal node containing node C. At node C, the ray spans its plane so the fragment is further clipped into two more ray fragments (see Figure 14.91).

The ray fragment in the back space of C is constructed from the original intersection point between the ray and node A and the intersection of the current fragment with node C. This is passed into the back node of C where it enters Leaf 3 and is added to the leaf collection list. The ray fragment in the front space of C is created from the intersection point of the ray fragment with the split plane of node C and the end point of the ray fragment passed into that function (the original ray end point).

So we have seen why the ray must be split at each node plane. This is a practice that we will see used again when working with BSP trees. Let us now look at the recursive function called by the CKDTree::CollectLeavesRay method to perform this ray splitting traversal through the tree.

Just like the ray traversal method for the other tree types, the parameters are the current node being visited, the leaf list for collection, and the ray origin and delta vectors.

```
bool CKDTree::CollectRayRecurse( CKDTreeNode * pNode,
                                 LeafList & List,
                                 const D3DXVECTOR3 & RayOrigin,
                                 const D3DXVECTOR3 & Velocity )
{
    CCollision::CLASSIFYTYPE PointA, PointB;
    D3DXVECTOR3 RayEnd, Intersection;
    bool        bResult;
    D3DXPLANE   Plane;
    float       t;

    // No operation if the node was null
    if ( !pNode ) return false;

    // If this node stores a leaf, just add it to the list and return
    if ( pNode->Leaf )
    {
        // Add the leaf to the list
        List.push_back( pNode->Leaf );

        // We collected a leaf
        return true;

    } // End if stores a leaf
```

The section of code above shows what happens when the node being visited is a leaf node. This can only happen if a fragment of the original ray has made it into a leaf. When this is the case the node's leaf pointer is simply added to the leaf collection list and the function returns true so that the caller knows that at least part of the ray was found to exist in at least one leaf.

The remainder of the code is executed only when a non-terminal node is entered. The first thing we do is add the ray delta vector to the ray origin to calculate the ray end position. We also use a local variable to point to the node plane for ease of access. We then classify the ray origin and ray end points against the split plane and store the classification results in the PointA and PointB local variables.

```
    // Calculate the end point of the ray
    RayEnd = RayOrigin + Velocity;

    // Retrieve the plane, and classify the ray points against it
    Plane  = pNode->Plane;

    PointA = CCollision::PointClassifyPlane( RayOrigin,
                                             (D3DXVECTOR3&)Plane,
                                             Plane.d );
```

```
    PointB = CCollision::PointClassifyPlane( RayEnd   ,
                                             (D3DXVECTOR3&)Plane,
                                             Plane.d );
```

If both points of the ray lay on the plane then the ray lies exactly on that plane. In this case, we will pass the ray into the front and back children. If a ray is on the plane, it is on the boundary of both nodes, so we should pass it into both nodes. Notice that we initially set the result boolean to false and set it true if any of the front and back recursions return true. We then return this boolean from the function so that the intersection status is returned back up to the root.

```
    // Test for the combination of ray point positions
    if ( PointA == CCollision::CLASSIFY_ONPLANE &&
         PointB == CCollision::CLASSIFY_ONPLANE )
    {
        // Traverse down the front and back
        bResult = false;
        if ( CollectRayRecurse( pNode->Front,
                                List,
                                RayOrigin,
                                Velocity ) ) bResult = true;

        if ( CollectRayRecurse( pNode->Back,
                                List,
                                RayOrigin,
                                Velocity ) ) bResult = true;
        return bResult;

    } // End If both points on plane
```

If the ray origin is in front of the plane and the ray end point is behind the plane (i.e., we have a spanning case), we perform a ray/plane intersection to calculate the *t* value from the ray origin. Once we have the *t* value we calculate the intersection point by scaling the ray delta vector by *t* and adding the result to the ray origin. Once we have the intersection point we recurs into the front child with the sub-ray constructed from the ray origin and the intersection point and recurs into the back child using the sub-ray constructed from the ray intersection point and the ray end point. If either function returns true, we set the boolean result to true and return.

```
    else

    if ( PointA == CCollision::CLASSIFY_INFRONT &&
         PointB == CCollision::CLASSIFY_BEHIND )
    {
        // The ray is spanning the plane, with the origin in front.
        CCollision::RayIntersectPlane( RayOrigin,
                                       Velocity,
                                       (D3DXVECTOR3&)Plane,
                                       Plane.d,
                                       t,
                                       true );

        Intersection = RayOrigin + (Velocity * t);

        // Traverse down both sides passing the relevant segments of the ray
```

```
        bResult = false;

        if ( CollectRayRecurse( pNode->Front,
                                List,
                                RayOrigin,
                                Intersection - RayOrigin ) ) bResult = true;

        if ( CollectRayRecurse( pNode->Back,
                                List,
                                Intersection,
                                RayEnd - Intersection ) ) bResult = true;
        return bResult;

    } // End If Spanning with origin in front
```

The next case is almost identical to the previous spanning case but is executed if the ray origin is behind the plane and the ray end point is in front of the plane. The same basic steps are taken, but the sub-ray fed into each halfspace is constructed differently due the different orientation of the plane with respect to the ray. This time, the sub-ray that we feed into the front node starts at the original ray end point and has the intersection point as its end. The ray we feed into the back child starts at the intersection point and has the original ray origin as its end point.

```
    else

    if ( PointA == CCollision::CLASSIFY_BEHIND &&
         PointB == CCollision::CLASSIFY_INFRONT )
    {
        // The ray is spanning the plane, with the origin in front.
        CCollision::RayIntersectPlane( RayOrigin,
                                       Velocity,
                                       (D3DXVECTOR3&)Plane,
                                       Plane.d,
                                       t,
                                       true );

        Intersection = RayOrigin + (Velocity * t);

        // Traverse down both sides passing the relevant segments of the ray
        bResult = false;

        if ( CollectRayRecurse( pNode->Front,
                                List,
                                Intersection,
                                RayEnd - Intersection ) ) bResult = true;

        if ( CollectRayRecurse( pNode->Back,
                                List,
                                RayOrigin,
                                Intersection - RayOrigin ) ) bResult = true;
        return bResult;

    } // End If Spanning with origin in front
```

The next cases are easier. If both points are in front of the plane, then the ray is totally contained in the plane's front space and we just pass the ray down the front child. Notice that we are actually saying that if Point A or Point B is in front of the plane, we send the entire ray down the front. Remember though, we only get here if the ray is *not* spanning the plane, so this tells us that if Point A or Point B is in the front space then the other must be in the front space too or on the plane (which also counts as the ray being fully contained in the front space).

```
    else

    if ( PointA == CCollision::CLASSIFY_INFRONT ||
         PointB == CCollision::CLASSIFY_INFRONT )
    {

        // Either of the points are in front (but not spanning), pass down front
        return CollectRayRecurse( pNode->Front, List, RayOrigin, Velocity );

    } // End if either point in front
```

Finally, the last code block only gets executed if the ray is behind the plane. That is, if one of the points is definitely behind the plane and the other is either behind the plane or situated on it. When this is the case we pass the entire ray down to the back child.

```
    else
    {
        // Either of the points are behind (but not spanning), pass down  back
        return CollectRayRecurse( pNode->Back, List, RayOrigin, Velocity );

    } // End if either point behind
}
```

Although the coverage of that function might have seemed more laborious than the simple box test, it was good practice to work with a recursive ray splitting procedure like this. We will see such ideas again when working with BSP trees in the coming lessons, so it was worth reviewing. Of course, functions like this can be hard to follow along with in your head (or even on paper) once the tree gets more than a few levels deep, but just remember that all we are doing here is a ray/plane test and creating two rays using the three points at our disposal (the ray origin, the ray intersection point, and the ray end point). We then pass these rays down to the children and eventually all fragments will pop out in the leaves in which they belong. Whenever a fragment enters a leaf, we add the leaf to the list and thus, track and return all leaves that are intersected by the ray.

14.20.4 The kD-Tree Debug Draw Routine

We will not spend time showing the code to the CKDTree::DebugDraw method even though it has been implemented in the source code. If you study the code you will see that it is basically identical to its quad-tree and oct-tree counterparts with the exception that it steps into two children rather than of four or eight.

This concludes our discussion and implementation of spatial trees for this lesson. We now have four tree types (quad-tree, YV quad-tree, oct-tree, and kD-tree) to choose from for use as a spatial manager in future applications. The remainder of this lesson will discuss the addition of a broad phase to our collision system using any of the ISpatialTree derived classes we introduced. As far the collision system (and our application) is concerned, these trees are all the same (they are ISpatialTrees) and thus can be used interchangeably without the user needing to understand the underlying data types.

14.21 Adding Spatial Trees to Lab Project 14.1

Lab Project 14.1 will include the core source code for this lesson and the next. Thus, much of the code in this project associated with the rendering system will be explained in the following lesson. In this lesson, we will focus our attention on how the spatial tree is populated with polygon data and how the collision system uses it to perform very efficient collision queries. Before we discuss the changes to the collision system, we will look at some functionality that has been added to other modules to support the introduction of spatial trees to our existing framework.

14.21.1 The CGameApp Class

Earlier in this lesson we discussed how our collision system will need to make sure that it does not send the same polygon into the expensive narrow phase multiple times. It avoids this by storing a value in each polygon as and when it processes it. As the value it stores is different every time a new query is issued, the collision system knows that if a polygon it is about to test has the same value stored in its structure, it must be one that has been processed already in this frame (perhaps it exists in a leaf that has been previously tested).

Although the system used will be fleshed out in much more detail when we discuss the change to the collision code, we have decided that this functionality might be useful for other modules as well. Therefore, we will store this value in CGameApp and expose two functions that return and increment this counter.

Excerpt from CGameApp.h

```
ULONG               GetAppCounter       () const { return m_nAppCounter; }
void                IncrementAppCounter ()       { m_nAppCounter++; }
```

The application counter is a simple ULONG that can be retrieved and incremented via the two methods shown above. Below we see that the current value of the application counter is stored in a new CGameApp member variable called m_nAppCounter.

Excerpt from CGameApp.h

```
ULONG               m_nAppCounter;      // Simple Application counter
```

Later you will see why it is useful to have an application global value that can be incremented and retrieved in this way.

14.21.2 The CScene Class

The CScene class has now had an ISpatialTree pointer added to its list of members. It is called m_pSpatialTree and will contain the pointer to the derived tree class we are currently using in our application. Because this is a pointer to a base class, we can make this point at any of our derived tree classes (CQuadTree, COctTree, etc.). Thus, we can easily plug in any of our tree classes without changing any code.

Excerpt from CScene.h

```
ISpatialTree       *m_pSpatialTree;                 // Spatial partitioning tree.
```

This pointer is set to NULL in CScene's constructor, as shown below.

Excerpt from CScene::CScene

```
m_pSpatialTree     = NULL;
```

LoadSceneFromIWF – CScene

The LoadSceneFromIWF function is the parent function that loads and processes the data loaded stored in an IWF file. We will not show all the code again here, so you may want to open the source file to check it out for yourself. What we will look at below is a shortened layout of this function just so you can see how the spatial tree is allocated, built, and registered with the collision system.

Excerpt from CScene::LoadSceneFromIWF

```
m_pSpatialTree = new COctTree( m_pD3DDevice, m_bHardwareTnL );

…
…
… Call Process Functions
…
…
…

if ( !m_pSpatialTree->Build( ) ) return false;

m_Collision.SetSpatialTree( m_pSpatialTree );
…
…
… Create Sound Manager Here
…
…
```

Near the top of the function we allocate a spatial tree of the type we wish to use. In this example, our COctTree class is selected as the spatial manager, but it could have been any one of our derived tree types.

The next section of the function loads the data and calls a series of processing functions to extract the information from the geometry and material vectors populated by the CFileIWF::Load method. One such function that has been changed slightly is the CScene::ProcessVertices method, which is called from CScene::ProcessMeshes (called from LoadSceneFromIWF) for each face loaded from the IWF file. The ProcessVertices method is passed the polygon (as an iwfSurface object) and stores its vertex data in a way that the application can parse and use. Our implementation of this function will create a new CPolygon and populate it with the vertex data for the face currently being loaded. It will then use the ISpatialTree::AddPolygon method to add that polygon to the spatial tree.

After the processing methods have been called, all static polygon data will have been added to the tree, so we issue a call to the tree's Build method, which as we now know, compiles the tree. After the Build function returns, our tree is ready to be used by the collision system, so we call the collision system's new SetSpatialTree method which stores a pointer to this tree for later use in collision queries. We will discuss how the collision system uses the spatial tree in a moment. The rest of the function is unchanged.

ProcessVertices - CScene

We have made some small changes in this function so that the vertex data for the passed face is stored in a new CPolygon structure and added to the spatial tree. Let us have a look at the code.

In the first section of the function we allocate a new CPolygon structure and set its normal to the normal of the passed face. We also set the attribute ID of the polygon to that passed into the function by ProcessMeshes. If the BackFace boolean is set to true, it means we would like to reverse the winding order of this polygon before we use it, so we negate the normal.

```
bool CScene::ProcessVertices( iwfSurface * pFilePoly,
                              ULONG nAttribID,
                              bool BackFace /* = false */ )
{
    // Validate parameters
    if ( !pFilePoly ) return false;

    long      i, nOffset = 0;
    CVertex * pVertices = NULL;
    float     fScale = 1.00f;

    // Allocate a new empty polygon
    CPolygon * pPolygon = new CPolygon;
    if ( !pPolygon ) return false;

    // Set the polygon's attribute ID
    pPolygon->m_nAttribID = nAttribID;
    pPolygon->m_vecNormal = (D3DXVECTOR3&)pFilePoly->Normal;

    if ( BackFace ) pPolygon->m_vecNormal = -pPolygon->m_vecNormal;
```

We then extract the number of vertices in the passed face and use this to inform the CPolygon to allocate enough space in its vertex array for the correct number of vertices.

```
    // Allocate enough vertices
    if ( pPolygon->AddVertex( pFilePoly->VertexCount ) < 0 ) return false;
    pVertices = pPolygon->m_pVertex;
```

Now we loop through each vertex and copy its data into the CPolygon vertex array.

```
    // If we are adding a back-face, setup the offset
    if ( BackFace ) nOffset = pFilePoly->VertexCount - 1;

    // Loop through each vertex and copy required data.
    for ( i = 0; i < (signed)pFilePoly->VertexCount; i++ )
    {
        // Copy over vertex data
        pVertices[i + nOffset].x      = pFilePoly->Vertices[i].x * fScale;
        pVertices[i + nOffset].y      = pFilePoly->Vertices[i].y * fScale;
        pVertices[i + nOffset].z      = pFilePoly->Vertices[i].z * fScale;

        pVertices[i + nOffset].Normal = D3DXVECTOR3&)pFilePoly->Vertices[i].Normal;

        if ( BackFace ) pVertices[i+nOffset].Normal=-pVertices[i + nOffset].Normal;

        // If we have any texture coordinates, set them
        if ( pFilePoly->TexChannelCount > 0 && pFilePoly->TexCoordSize[0] == 2 )
        {
            pVertices[i + nOffset].tu = pFilePoly->Vertices[i].TexCoords[0][0];
            pVertices[i + nOffset].tv = pFilePoly->Vertices[i].TexCoords[0][1];

        } // End if has tex coordinates

        // If we're adding the backface, decrement the offset
        // Remember, we decrement by two here because 'i' will increment
        if ( BackFace ) nOffset -= 2;

    } // Next Vertex
```

Now that we have a CPolygon representation of the face loaded from the file, we add it to the spatial tree.

```
    // Add this new polygon to the spatial tree
    if ( !m_pSpatialTree->AddPolygon( pPolygon ) ) {delete pPolygon;
                                                    return false; }

    // Success!
    return true;
}
```

These are the only changes to the application framework with respect to populating and building the tree. Next we will discuss the changes to the collision system that need to be made so that we finally have a broad phase in our collision detection code.

14.22 Adding a Broad Phase to CCollision

The collision system will not require significant changes in order to include a broad phase. Most of the changes will involve the addition of new functions to collect tree data. You will recall from the previous chapter that the application calls the CCollision::CollideEllipsoid method to perform a collision query. This method transformed the passed ellipsoid into eSpace and then entered a loop that repeatedly called the EllipsoidIntersectScene method to perform collision detection. If the function returned false, then the ellipsoid could move to the requested position, otherwise a response vector was generated and the collision detection step was called again.

Inside the EllipsoidIntersectScene method is where the detection between the swept sphere and the collision geometry was determined. Detection was performed against three pools of geometry using the EllipsoidIntersectBuffers method for the core intersection determination in all three cases. The first pool we tested was any terrain geometry that may have been registered. We did not store the terrain triangles in the collision system's static geometry array, but instead generated them on the fly using a temporary buffer that was passed to the EllipsoidIntersectBuffers method. After recording any intersections with terrain triangles, we then iterated through the collision system's dynamic object array passing the geometry buffers of each dynamic object into the EllipsoidIntersectBuffers method to record and store any collisions that occurred. Finally, we passed the collision system's static geometry buffers into the EllipsoidIntersectBuffers method so that the static polygon data used by the system was also tested and any collisions recorded.

The EllipsoidIntersectScene method will now be updated in two places. First, there will be a fourth pool of geometry that needs to be checked -- the geometry in the spatial tree that has been assigned to the collision system. As we have seen, the application populates and builds this tree using static scene data and then assigns it to the tree using the CCollision::SetSpatialTree method (which just caches the pointer in a member variable). During the EllipsoidIntersectScene method, we will test the geometry in the tree by sending the AABB of the swept sphere into the spatial tree's CollectLeavesAABB method. Once all intersecting leaves are returned, we will test each polygon in those leaves to see if their bounding boxes intersect the bounding box of the swept sphere (an additional broad phase step). We will only perform narrow phase intersection for polygons that have a good potential for intersect the sphere's AABB. This is essentially our broad phase implementation for static geometry.

One thing might be bothering you at this point. If the static polygon data that we load will now be stored in the tree, what happens to the original static geometry database in the collision system? Well, those vectors will no longer be used because we will always choose to use a spatial tree and connect that to the collision system. We could still add geometry to the collision system's static vectors (using the CCollision::AddIndexedPrimitive method for example) but as we know, every polygon in these vectors will be tested in the narrow phase. Thus, storing static geometry there instead of in our spatial tree would force us to give up our extremely efficient broad phase step. So does that mean that all of the functions that we wrote to store polygons and collide against the static geometry were a waste of time? Absolutely not. Our aim was to create a collision library that can be used in your applications that might rely on other technologies. Although we will favor using a spatial tree to store our static geometry, there may well be users that wish to use the collision system without those trees. Therefore, while it is recommended that you always use a spatial tree to store your static collision data, the collision system

will still do its job even if one is not provided. As we saw in the previous lesson, if somebody just wants to quickly add polygon data to the collision system, we have provided a means to do so.

The other area where the EllipsoidIntersectScene method will change slightly will when testing the swept sphere against dynamic objects. In the previous chapters we saw that we did this by passing the geometry buffers of the dynamic object currently being processed (and its matrix) into the EllipsoidIntersectBuffers method. This essentially performed the narrow phase detection step on every polygon in the dynamic object's buffer. Now, we first test the bounding box of the dynamic object against the AABB of the swept sphere and only call the EllipsoidIntersectBuffers method if the two intersect. Although this may seem like a very crude broad phase for our dynamic objects, AABB/AABB intersection tests are so cheap and our project uses so few dynamic objects there is really no point in using a tree traversal to determine which objects intersect. Therefore, unlike the rendering system that we will cover in the following lesson, our collision system does not use the spatial tree to implement a broad phase for dynamic objects, instead it just loops through each performing an AABB/AABB cull.

> **Note:** It would be easy enough to add some code to take advantage of the tree for your dynamic objects. After all, they are already stored in the tree and you would already have incurred the cost of traversal when you test for static geometry. Thus, there would be only a few extra lines of logic needed to extract the dynamic objects from the leaves you collided with.

In order to implement the AABB/AABB broad phase for the dynamic object testing we will need to extend our collision system's DynamicObject structure to make room for two vectors that will store the minimum and maximum extents of its bounding box. Remembering from the previous lesson that dynamic objects are stored in the collision system as model space geometry with an associated world transformation matrix, the bounding box stored in each dynamic object will also be defined in the model space of the object too. We will see later that when performing the AABB/AABB test between the swept sphere's AABB and the dynamic object, the dynamic object's AABB will be transformed into world space for the test.

Our updated DynamicObject structure is shown below with the two new members highlighted in bold.

Excerpt from CCollision.h

```
struct DynamicObject
{
        CollTriVector  *pCollTriangles;
        CollVertVector *pCollVertices;
        D3DXMATRIX     *pCurrentMatrix;
        D3DXMATRIX      LastMatrix;
        D3DXMATRIX      VelocityMatrix;
        D3DXMATRIX      CollisionMatrix;
        long            ObjectSetIndex;
        bool            IsReference;
        D3DXVECTOR3     BoundsMin;      // Minimum bounding box extents
        D3DXVECTOR3     BoundsMax;      // Maximum bounding box extents
};
```

We just learned that the EllipsoidIntersectScene method will now perform a collision query on the tree. That is, it will use the spatial tree's CollectLeavesAABB method to return a list of all the leaves intersected by the AABB of the swept sphere. Two new members have been added to the CCollision

class for this purpose. The first is a pointer to an ISpatialTree derived object which the application will set using the CCollision::SetSpatialTree. The second is a member of type ISpatialTree::LeafList. Remember, when we run an AABB query on the tree, the CollectLeavesAABB method expects to be passed an empty leaf list that it will fill with leaf pointers. Therefore, the collision object maintains a leaf list of its own which it can empty each time and pass into this function to collect the leaves whenever a query is performed. Thus, once our collision system has called the ISpatialTree::CollectLeavesAABB method, its m_treeLeafList member will contain a list of all the leaves intersected by the swept sphere's AABB. The new members of CCollision are shown below.

New Members to CCollision

```
ISpatialTree           *m_pSpatialTree;
ISpatialTree::LeafList  m_TreeLeafList;
D3DXVECTOR3             m_vecEllipsoidMin;
D3DXVECTOR3             m_vecEllipsoidMax;
```

Notice that we have also added two new 3D vectors as members in the collision object. Because we will need to access the world space bounding box of the ellipsoid during our EllipsoidIntersectScene method, we use these new members to cache the extents. For example, we will need the world space AABB of the ellipsoid in the methods that send the AABB of the ellipsoid down the tree to collect all intersecting leaves. We will also need the world space AABB during the dynamic object tests when we implement a broad phase AABB/AABB test between the world space AABB of the dynamic object and the world space AABB of the ellipsoid. Rather than continually having to calculate this box each time we need it, we will add a function to calculate the ellipsoid bounds. It will generate a bounding box around the ellipsoid and store it in these two variables for the life of the query. This will allow any helper functions that need this information to be able to quickly access it.

SetSpatialTree - CCollision

One of the new methods that we saw being used earlier assigns the collision object's m_pSpatialTree pointer the tree the application would like it to use. This simple function is called once the application has created and populated the tree with the required data.

```
void CCollision::SetSpatialTree( ISpatialTree * pTree )
{
    // Store the tree
    m_pSpatialTree = pTree;
}
```

CalculateEllipsoidBounds – CCollision

This method calculates the bounding box of the query ellipsoid. It tests the x, y, and z components of two points, the ellipsoid center position and the ellipsoid center position plus the velocity vector, to find the minimum and maximum extents for a bounding box that encompasses both points. We will see this method being used in a few of the other collision functions later on.

The function is passed the ellipsoid center position, its radius vector, and its velocity vector. It first examines the radius vector to determine which of its components is the largest. We will use this extent to essentially represent a sphere that bounds the ellipsoid. Notice at the top of the function how we reference the member variables m_vecEllipsoidMin and m_vecEllipsoidMax (which will receive the resulting AABB) with some local variables for ease of access.

```
void CCollision::CalculateEllipsoidBounds( const D3DXVECTOR3& Center,
                                           const D3DXVECTOR3& Radius,
                                           const D3DXVECTOR3& Velocity )
{
    float       fLargestExtent;

    D3DXVECTOR3& vecMin = m_vecEllipsoidMin;
    D3DXVECTOR3& vecMax = m_vecEllipsoidMax;

    // Find largest extent of ellipsoid
    fLargestExtent = Radius.x;
    if ( Radius.y > fLargestExtent ) fLargestExtent = Radius.y;
    if ( Radius.z > fLargestExtent ) fLargestExtent = Radius.z;
```

At this point we have stored the largest component of the ellipsoid's radius vector in fLargestExtent. We will now initialize the bounding box to impossibly small values before performing the tests.

```
    // Reset the bounding box values
    vecMin = D3DXVECTOR3( FLT_MAX, FLT_MAX, FLT_MAX );
    vecMax = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX );
```

The first thing we will do is add the largest extent to the ellipsoid center position and test the result to see if it is greater than the value currently stored in the bounding box maximum extent. We do this on a per-component basis. We are essentially growing the box if the sphere that surrounds the ellipsoid pierces the previously discovered maximum extent along any axis. If it does, that component of the maximum extent is updated with the new maximum.

```
    // Calculate the bounding box extents of where the ellipsoid currently
    // is, and the position it will be moving to.

    if ( Center.x + fLargestExtent > vecMax.x )
      vecMax.x = Center.x + fLargestExtent;

    if ( Center.y + fLargestExtent > vecMax.y )
      vecMax.y = Center.y + fLargestExtent;

    if ( Center.z + fLargestExtent > vecMax.z )
      vecMax.z = Center.z + fLargestExtent;
```

We now do exactly the same test for each component to test that the sphere bounding the ellipsoid does not pierce any previously found minimum extent for the box along any world axis. Once again, if it does, that component of the box is updated to the new minimum.

```
    if ( Center.x - fLargestExtent < vecMin.x )
```

```
      vecMin.x = Center.x - fLargestExtent;

    if ( Center.y - fLargestExtent < vecMin.y )
      vecMin.y = Center.y - fLargestExtent;

    if ( Center.z - fLargestExtent < vecMin.z )
      vecMin.z = Center.z - fLargestExtent;
```

Now repeat the same two tests again, only this time we add the velocity vector and the largest radius extent to the center position of the ellipsoid. What we are essentially doing is building a sphere that bounds the ellipsoid at the very end of its velocity vector.

```
    if ( Center.x + Velocity.x + fLargestExtent > vecMax.x )
      vecMax.x = Center.x + Velocity.x + fLargestExtent;

    if ( Center.y + Velocity.y + fLargestExtent > vecMax.y )
      vecMax.y = Center.y + Velocity.y + fLargestExtent;

    if ( Center.z + Velocity.z + fLargestExtent > vecMax.z )
      vecMax.z = Center.z + Velocity.z + fLargestExtent;

    if ( Center.x + Velocity.x - fLargestExtent < vecMin.x )
      vecMin.x = Center.x + Velocity.x - fLargestExtent;

    if ( Center.y + Velocity.y - fLargestExtent < vecMin.y )
      vecMin.y = Center.y + Velocity.y - fLargestExtent;

    if ( Center.z + Velocity.z - fLargestExtent < vecMin.z )
      vecMin.z = Center.z + Velocity.z - fLargestExtent;
```

Finally, just to add a bit of padding to manage limited floating point precision, we extend the box by one unit along each axis in both directions.

```
    // Add Tolerance values
    vecMin -= D3DXVECTOR3( 1.0f, 1.0f, 1.0f );
    vecMax += D3DXVECTOR3( 1.0f, 1.0f, 1.0f );
}
```

We will see this method being used later when discussing the new additions to the collision system.

14.22.1 Dynamic Object Bounding Box Generation

When a dynamic object is registered with the collision system, it will now have to calculate its model space bounding box so that it can be stored in the DynamicObject structure and used during the dynamic object's broad phase test. Adding such a feature involves simply extending the code of the CCollision::AddBufferData method.

You should recall from the previous chapter how the AddBufferData method was a general purpose utility function that was used to add a passed array of vertices and indices to the passed geometry

vectors. This was useful because the same function could be used to add the vertex data for a static polygon to the collision system's static geometry vectors, or it could be used to add the vertex and index data for a dynamic object to the dynamic object's geometry buffers. The AddBufferData method could even be passed an optional matrix pointer parameter that would be used to transform the vertex data prior to storing it in the passed geometry buffers.

To jog your memory, below we see an example of how this method was used by the CCollision::AddIndexedPrimitive method. This method was called in our previous application for every static polygon loaded from the IWF file. As you can see, this method just issues a call to the AddBufferData method, passing as the first two parameters the static geometry buffers of the collision system. When the function returns, the passed vertices and indices will have been added to the collision system's static vectors. We are also reminded that the current transformation matrix for the collision system is also passed in, so that if the application has set this to anything other than an identity matrix (its default state) the geometry will be transformed before it is added to the collision system's static geometry vectors.

```
bool CCollision::AddIndexedPrimitive( LPVOID Vertices,
                                      LPVOID Indices,
                                      ULONG VertexCount,
                                      ULONG TriCount,
                                      ULONG VertexStride,
                                      ULONG IndexStride,
                                      USHORT MaterialIndex )
{
    ULONG i, BaseTriCount;

    // Store the previous triangle count
    BaseTriCount = m_CollTriangles.size();

    // Add to the standard buffer
    if ( !AddBufferData( m_CollVertices,
                         m_CollTriangles,
                         Vertices,
                         Indices,
                         VertexCount,
                         TriCount,
                         VertexStride,
                         IndexStride,
                         m_mtxWorldTransform ) ) return false;

    // Loop through and assign the specified material ID to all triangles
    for ( i = BaseTriCount; i < m_CollTriangles.size(); ++i )
    {
        // Assign to triangle
        m_CollTriangles[i].SurfaceMaterial = MaterialIndex;

    } // Next Triangle

    // Success
    return true;
}
```

The AddBufferData method was also used to add dynamic objects to the system. In this case, the vertex and index data of the object would be passed as the first two parameters, along with the geometry buffers for the dynamic object. The vertex and index data would be copied into the geometry buffers of the collision system's representation of that dynamic object as a result.

Because our dynamic objects will now need to have bounding boxes calculated for them, the AddBufferData member has had two optional parameters added which are only used when the function is being called for dynamic objects. These two parameters are pointers to the dynamic object's bounding box extents. The function will calculate the bounding box extents and store them in the dynamic object before returning.

Below we see the AddDynamicOject method of CCollision which the application can use to register a single dynamic object with the collision system. This code is virtually unchanged from the previous version with the exception that it now passes the addresses of the dynamic object's bounding box extents as the last two parameters to AddBufferData. All of the other code was discussed in detail over the last two lessons, and will not be discussed here.

```
long CCollision::AddDynamicObject( LPVOID Vertices,
                                   LPVOID Indices,
                                   ULONG VertexCount,
                                   ULONG TriCount,
                                   ULONG VertexStride,
                                   ULONG IndexStride,
                                   D3DXMATRIX * pMatrix,
                                   bool bNewObjectSet /* = true */ )
{
    D3DXMATRIX      mtxIdentity;
    DynamicObject * pObject = NULL;

    // Reset identity matrix
    D3DXMatrixIdentity( &mtxIdentity );

    // Ensure that they aren't doing something naughty
    if ( m_nLastObjectSet < 0 ) bNewObjectSet = true;

    // We have used another object set index.
    if ( bNewObjectSet ) m_nLastObjectSet++;

    try
    {
        // Allocate a new dynamic object instance
        pObject = new DynamicObject;
        if ( !pObject ) throw 0;

        // Clear the structure
        ZeroMemory( pObject, sizeof(DynamicObject) );

        // Allocate an empty vector for the buffer data
        pObject->pCollVertices  = new CollVertVector;
        if ( !pObject->pCollVertices ) throw 0;
        pObject->pCollTriangles = new CollTriVector;
        if ( !pObject->pCollTriangles ) throw 0;
```

```
        // Store the matrices we'll need for
        pObject->pCurrentMatrix  = pMatrix;
        pObject->LastMatrix      = *pMatrix;
        pObject->CollisionMatrix = *pMatrix;
        pObject->ObjectSetIndex  = m_nLastObjectSet;
        pObject->IsReference     = false;
        D3DXMatrixIdentity( &pObject->VelocityMatrix );

        // Add to the dynamic object's database
        if ( !AddBufferData( *pObject->pCollVertices,
                            *pObject->pCollTriangles,
                             Vertices,
                             Indices,
                             VertexCount,
                             TriCount,
                             VertexStride,
                              IndexStride,
                              mtxIdentity,
                              &pObject->BoundsMin,
                              &pObject->BoundsMax ) ) throw 0;

        // Store the dynamic object
        m_DynamicObjects.push_back( pObject );

    } // End try block

    catch (...)
    {
        // Release the object if it got created
        if ( pObject )
        {
            if ( pObject->pCollVertices  ) delete pObject->pCollVertices;
            if ( pObject->pCollTriangles ) delete pObject->pCollTriangles;
            delete pObject;

        } // End if object exists

        // Return failure
        return -1;

    } // End Catch Block

    // Return the object index used
    return m_nLastObjectSet;
}
```

When AddBufferData returns, it will have calculated the extents of the model space AABB and will have stored them in the final two parameters.

You will see this slight modification to the AddBufferData call in all of the methods we discussed which register dynamic objects. You will recall for example that when we register an actor with the collision system (using the CCollision::AddActor method) and pass the parameter that states that we would like the actor to be dynamic, the hierarchy is traversed and a dynamic object is created in the collision

system for each mesh container found. However, each dynamic object created from a mesh container is created in exactly the same way as shown above and will now have their bounding boxes computed.

The new version of AddBufferData is shown below with the added code that calculates and stores the model space AABB of the passed geometry.

The first section of the function is the same, it simply makes room in the passed vertex and triangle buffers to add the new data if the buffers are not currently large enough.

```
bool CCollision::AddBufferData( CollVertVector& VertBuffer,
                                CollTriVector& TriBuffer,
                                LPVOID Vertices,
                                LPVOID Indices,
                                ULONG VertexCount,
                                ULONG TriCount,
                                ULONG VertexStride,
                                ULONG IndexStride,
                                const D3DXMATRIX& mtxWorld,
                                D3DXVECTOR3 * pBoundsMin /* = NULL */,
                                D3DXVECTOR3 * pBoundsMax /* = NULL */ )
{
    ULONG        i, Index1, Index2, Index3, BaseVertex;
    UCHAR       *pVertices = (UCHAR*)Vertices;
    UCHAR       *pIndices  = (UCHAR*)Indices;
    CollTriangle Triangle;
    D3DXVECTOR3  Edge1, Edge2;

    // Validate parameters
    if ( !Vertices || !Indices || !VertexCount || !TriCount ||
         !VertexStride || (IndexStride != 2 && IndexStride != 4) ) return false;

    // Catch template exceptions
    try
    {
        // Grow the vertex buffer if required
        while ( VertBuffer.capacity() < VertBuffer.size() + VertexCount )
        {
            // Reserve extra space
            VertBuffer.reserve( VertBuffer.capacity() + m_nVertGrowCount );

        } // Keep growing until we have enough

        // Grow the triangle buffer if required
        while ( TriBuffer.capacity() < TriBuffer.size() + TriCount )
        {
            // Reserve extra space
            TriBuffer.reserve( TriBuffer.capacity() + m_nTriGrowCount );

        } // Keep growing until we have enough
```

Next we store the current number of vertices in the passed vertex vector so that we know where the indices will start indexing (it might not be empty when the function is called, which is certainly the case

when the function is called to add triangles to the static geometry vectors). We also initialize the passed extent vectors to impossible values.

```
        // Store the original vertex count before we copy
        BaseVertex = VertBuffer.size();

        // Reset bounding box values if provided
        if ( pBoundsMin ) *pBoundsMin = D3DXVECTOR3( FLT_MAX, FLT_MAX, FLT_MAX );
        if ( pBoundsMax ) *pBoundsMax = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX);
```

Now we loop through each vertex in the passed array and transform it by the passed transformation matrix. We then test the current bounding box extents against the vertex and grow the box if the vertex is found not to be contained in it. We then add the vertex to the passed vertex vector.

```
        // For each vertex passed
        for ( i = 0; i < VertexCount; ++i, pVertices += VertexStride )
        {
            // Transform the vertex
            D3DXVECTOR3 Vertex = *(D3DXVECTOR3*)pVertices;
            D3DXVec3TransformCoord( &Vertex, &Vertex, &mtxWorld );

            // Calculate bounding box extents
            if ( pBoundsMin )
            {
                if ( Vertex.x < pBoundsMin->x ) pBoundsMin->x = Vertex.x;
                if ( Vertex.y < pBoundsMin->y ) pBoundsMin->y = Vertex.y;
                if ( Vertex.z < pBoundsMin->z ) pBoundsMin->z = Vertex.z;

            } // End if minimum extents variable passed

            if ( pBoundsMax )
            {
                if ( Vertex.x > pBoundsMax->x ) pBoundsMax->x = Vertex.x;
                if ( Vertex.y > pBoundsMax->y ) pBoundsMax->y = Vertex.y;
                if ( Vertex.z > pBoundsMax->z ) pBoundsMax->z = Vertex.z;

            } // End if maximum extents variable passed

            // Copy over the vertices
            VertBuffer.push_back( Vertex );

        } // Next Vertex
```

The last section of the function calculates the indices of each triangle and is unchanged.

```
        // Build the triangle data
        for ( i = 0; i < TriCount; ++i )
        {
            // Retrieve the three indices
            Index1 = ( IndexStride == 2 ? (ULONG)*((USHORT*)pIndices) :
                                          *((ULONG*)pIndices) );
            pIndices += IndexStride;

            Index2 = ( IndexStride == 2 ? (ULONG)*((USHORT*)pIndices) :
```

```
                                        *((ULONG*)pIndices) );
            pIndices += IndexStride;

            Index3 = ( IndexStride == 2 ? (ULONG)*((USHORT*)pIndices) :
                       *((ULONG*)pIndices) );
            pIndices += IndexStride;

            // Store the details
            Triangle.SurfaceMaterial = 0;
            Triangle.Indices[0]      = Index1 + BaseVertex;
            Triangle.Indices[1]      = Index2 + BaseVertex;
            Triangle.Indices[2]      = Index3 + BaseVertex;

            // Retrieve the vertices themselves
            D3DXVECTOR3 &v1 = VertBuffer[ Triangle.Indices[0] ];
            D3DXVECTOR3 &v2 = VertBuffer[ Triangle.Indices[1] ];
            D3DXVECTOR3 &v3 = VertBuffer[ Triangle.Indices[2] ];

            // Calculate and store edge values
            D3DXVec3Normalize( &Edge1, &(v2 - v1) );
            D3DXVec3Normalize( &Edge2, &(v3 - v1) );

            // Skip if this is a degenerate triangle, we don't want it in our set
            float fDot = D3DXVec3Dot( &Edge1, &Edge2 );
            if ( fabsf(fDot) >= (1.0f - 1e-5f) ) continue;

            // Generate the triangle normal
            D3DXVec3Cross( &Triangle.Normal, &Edge1, &Edge2 );
            D3DXVec3Normalize( &Triangle.Normal, &Triangle.Normal );

            // Store the triangle in our database
            TriBuffer.push_back( Triangle );

        } // Next Triangle

    } // End Try Block

    catch ( ... )
    {
        // Just fail on any exception.
        return false;

    } // End Catch Block


    // Success!
    return true;
}
```

That covers all of the changes involved in registering geometry with the collision system. They are very small, but the calculation of the bounding boxes of dynamic objects is a significant step that should not be overlooked. We can now move on to discussing the changes to the core collision system.

14.22.2 Changes to the Collision Query Engine

When the application performs a query, it calls the CCollision::CollideEllipsoid method. This method is responsible for transforming the ellipsoid into eSpace and then setting up a loop that will call the detection method (EllipsoidIntersectScene) and generate a slide vector every time the detection function returns true for intersection. Only when the detection phase returns false have we found a new position for the ellipsoid that is free from obstruction.

As we have briefly touched on, we need to make sure during a given detection step that we do not send a polygon to the expensive narrow phase more than once. If we are using a clipped tree, this will never happen since each polygon will be contained in one and only one leaf. However, if a non-clipped tree is being used (often the case) we may have a situation where a single polygon is contained in multiple leaves. Further, when the collision system calls the ISpatialTree::CollectLeavesAABB method, it may get back a list of intersecting leaves that each contains this same polygon. We certainly never want to test the same polygon more than once, so we saw earlier how we implemented an application timer to allow us to efficiently invalidate polygons in the collision system.

The CPolygon structure stores a member called m_AppCounter that can be used to store the current value held in the application counter. Why is this useful?

When our collision system performs a query on the tree, it will get back a list of leaves. Then it will iterate through the polygons in each of those leaves and potentially pass them on to the narrow phase if we have an AABB/AABB intersection between the polygon's box and the swept sphere's box. Once a polygon has been tested and is about to be passed to the narrow phase, we will store the current value of the application counter in its m_nAppCounter member. We will only send polygons to the narrow phase whose m_nAppCounter member is not equal to the current value of the application counter. If the application counter matches, it means this polygon has already been processed since the application counter was last incremented. This would be the case if we were processing a polygon in a leaf that had existed in a previous leaf we had tested. By the end of the collision step, all the polygons in all the returned leaves will have had their m_nAppCounter members set to the same value as the current application counter.

While we could achieve the same end by storing a boolean in each polygon which is set to true once it has been processed, this would leave us with the very unpleasant (and slow) task of looping through every polygon and resetting their booleans prior to performing a new query. Since our CollideEllipsoid method may perform many detection queries during a single update, it would become prohibitively expensive if tens of thousands of polygons had to have their booleans reset dozens of times. Fortunately, with our counting logic, all we have to do before performing another detection query is increment the application's main counter. At this point, the application counter will no longer match the m_nAppCounter member in any polygon, so they will all be tested in the next update.

Of you open up the source code to the CCollision::CollideEllipsoid method you will see that we have added a new function call that increments the application counter with each iteration of the detection loop. We will not show the entire function here since we have only added one new line. However, the

following listing demonstrates where this new line (actually two new lines which are both the same) exists. The new lines are highlighted in bold.

Excerpt from CCollision::CollideEllipsoid

```
bool CCollision::CollideEllipsoid( const D3DXVECTOR3& Center,
                                   const D3DXVECTOR3& Radius,
                                   const D3DXVECTOR3& Velocity,
                                   D3DXVECTOR3& NewCenter,
                                   D3DXVECTOR3& NewIntegrationVelocity,
                                   D3DXVECTOR3& CollExtentsMin,
                                   D3DXVECTOR3& CollExtentsMax )
{
      …
      … Calculate ESpace ellipsoid start, end and radius vectors here
      …
      …
    // Keep testing until we hit our max iteration limit
    for ( i = 0; i < m_nMaxIterations; ++i )
    {

        if ( EllipsoidIntersectScene( eFrom,
                                      Radius,
                                      eVelocity,
                                      m_pIntersections,
                                      IntersectionCount,
                                      true,
                                      true ) )
        {
            …
            … Calculate Response vector and eFrom here
            …
            bHit=true;

        } // End if we got some intersections
        else
        {
            // We found no collisions, so break out of the loop
            break;

        } // End if no collision

        // Increment the app counter so that our polygon testing is reset
        GetGameApp()->IncrementAppCounter();

    } // Next Iteration

    // Did we register any intersection at all?
    if ( bHit )
    {
        // Increment the app counter so that our polygon testing is reset
        GetGameApp()->IncrementAppCounter();

        …
        …
        …  If we ran out of iterations to one collision test here and just
        …  return first intersecting position
```

```
        …


    } // End if intersection found

    // Store the resulting output values
    NewCenter              = vecOutputPos;
    NewIntegrationVelocity = vecOutputVelocity;

    // Return hit code
    return bHit;
}
```

As you can see, the application counter is advanced after each collision detection query so that the next time it is performed, the polygon counters no longer match the application counter and the polygons are considered again in the next test.

EllipsoidIntersectScene - CCollision

The EllipsoidIntersectScene method is called by CollideEllipsoid each time a detection test has to be performed on a velocity vector (or slide vector). Previously this method tested three pools of data; the terrain data, the dynamic objects, and the static geometry vectors. We will now add an additional section that performs a collision test on the static geometry stored in the spatial tree (if one has been assigned to the collision system). In our application, all static data will be stored in the spatial tree and the static geometry vectors of the collision system will be left empty. We also add the AABB/AABB broad phase to the dynamic object test. Although most of the code is unchanged, we will still show the entire function again, but only spend time discussing the additional code in any detail.

The first section of the function transforms the passed ellipsoid centerpoint and velocity vector into eSpace if the caller did not pass these in as eSpace parameters to begin with. It also sets the first collision intersect time (*t*) to 1.0 (i.e., the end of the vector and thus no collision detected).

```
bool CCollision::EllipsoidIntersectScene( const D3DXVECTOR3 &Center,
                                          const D3DXVECTOR3& Radius,
                                          const D3DXVECTOR3& Velocity,
                                          CollIntersect Intersections[],
                                          ULONG & IntersectionCount,
                                          bool bInputEllipsoidSpace /* = false */,
                                          bool bReturnEllipsoidSpace /* = false */)
{
    D3DXVECTOR3 eCenter, eVelocity, eAdjust, vecEndPoint, InvRadius;
    float       eInterval;
    ULONG       i;

    // Vectors for terrain building
    CollVertVector VertBuffer;
    CollTriVector  TriBuffer;

    // Calculate the reciprocal radius to prevent the divides we would need
    InvRadius = D3DXVECTOR3( 1.0f / Radius.x, 1.0f / Radius.y, 1.0f / Radius.z );
```

```
    // Convert the values specified into ellipsoid space if required
    if ( !bInputEllipsoidSpace )
    {
        eCenter   = Vec3VecScale( Center, InvRadius );
        eVelocity = Vec3VecScale( Velocity, InvRadius );

    } // End if the input values were not in ellipsoid space
    else
    {
        eCenter   = Center;
        eVelocity = Velocity;

    } // End if the input values are already in ellipsoid space

    // Reset ellipsoid space interval to maximum
    eInterval = 1.0f;

    // Reset initial intersection count to 0 to save the caller having to do this.
    IntersectionCount = 0;

    // Calculate the bounding box of the ellipsoid
    D3DXVECTOR3 vecCenter   = Vec3VecScale( eCenter, Radius );
    D3DXVECTOR3 vecVelocity = Vec3VecScale( eVelocity, Radius );
    CalculateEllipsoidBounds( vecCenter, Radius, vecVelocity );
```

The new additions to this section are highlighted in bold at the bottom of the code. We transform the ellipsoid center position and velocity vector into world space (by scaling it by the radius vector) and then pass the world space position, velocity, and radius vector of the ellipsoid into the CalculateEllipsoidBounds method (discussed earlier). When this method returns, the CCollision::m_vecEllipsoidMin and CCollision::m_vecEllipsoidMax member variables will store the extents of the current world space bounding box of the ellipsoid. We will see why we need this later in the function.

The next section, which is unchanged from the previous lab project, collects triangles from any terrain object in the immediate vicinity of the swept sphere and runs the intersection test.

```
    // Iterate through our terrain database
    TerrainVector::iterator TerrainIterator = m_TerrainObjects.begin();
    for ( ; TerrainIterator != m_TerrainObjects.end(); ++TerrainIterator )
    {
        const CTerrain * pTerrain = *TerrainIterator;

        // Collect the terrain triangle data
        if ( !CollectTerrainData( *pTerrain,
                                  vecCenter,
                                  Radius,
                                  vecVelocity,
                                  VertBuffer,
                                  TriBuffer ) ) continue;

        // Perform the ellipsoid intersect test against this set of terrain data
        EllipsoidIntersectBuffers( VertBuffer,
                                   TriBuffer,
```

```
                                    eCenter,
                                    Radius,
                                    InvRadius,
                                    eVelocity,
                                    eInterval,
                                    Intersections,
                                    IntersectionCount );

        // Clear buffers for next terrain
        VertBuffer.clear();
        TriBuffer.clear();

    } // Next Terrain
```

The next section is new. It contains the code that performs the intersection test against the geometry stored in the spatial tree (if one has been registered with the collision system). The first thing we do is pass the world space ellipsoid center position and velocity vector to a new method called CCollision::CollectTreeLeaves. This method will simply pass the world space bounding box of the ellipsoid into the ISpatialTree::CollectLeavesAABB method. Earlier we learned that the collision system now includes a member of type LeafList, which is also passed to the CollectLeavesAABB method and used to collect the pointers to the leaves that intersect the ellipsoid AABB. We will look at this method in a moment.

When the CollectTreeLeaves method returns, the CCollision::m_TreeLeafList member will currently contain all the pointers to all leaves that intersect the swept sphere's bounding box. We then call another new method called EllipsoidIntersectTree which performs the narrow phase test on the polygon data stored in those leaves. The EllipsoidIntersectTree method is almost identical to the EllipsoidIntersectBuffers method which performs narrow phase detection on other geometry pools. The reason we had to make a special version of this function is that the CPolygon data stored in the tree uses a different format from what the collision system is used to working with. Rather than incur the costs of copying the data over and converting it into the old format on the fly (which can get very expensive!), we just added a modified version of the prior code to work with CPolygons. We only want to test polygons that have bounding boxes that intersect the ellipsoid's bounding box and that have not already been tested in this update. This is another method we will look at in a moment.

```
    // Collide against tree if available
    if ( m_pSpatialTree )
    {

        // Collect the leaf list if necessary
        CollectTreeLeaves( vecCenter, Radius, vecVelocity );

        EllipsoidIntersectTree( eCenter,
                                Radius,
                                InvRadius,
                                eVelocity,
                                eInterval,
                                Intersections,
                                IntersectionCount );

    } // Next Tree
```

When the EllipsoidIntersectTree method returns, any intersections found between the swept sphere and the polygons stored in the tree will be added to the passed Intersections array and the current intersection count will be increased to reflect the number of intersections found at the closest *t* value. That is the full extent of the changes to EllipsoidIntersectScene with respect to using the spatial tree to query the polygon data as a broad phase.

The next change to the function is performed when testing the dynamic objects against the swept sphere. In our previous implementation we simply looped through every dynamic object and passed its buffers to the EllipsoidIntersectBuffers method. This essentially meant that every polygon in every dynamic object would have been run through the costly narrow phase. Now we will add a simple broad phase early-out test.

For each dynamic object, we transform its model space bounding box into world space using a new math utility function called TransformAABB (see MathUtility.cpp). This is a small function that takes an AABB and a transformation matrix and generates a new AABB that contains the passed AABB in its rotated/transformed form. We will look at the code to this function later.

```
    // Iterate through our triangle database
    DynamicObjectVector::iterator ObjIterator = m_DynamicObjects.begin();
    for ( ; ObjIterator != m_DynamicObjects.end(); ++ObjIterator )
    {
        DynamicObject * pObject = *ObjIterator;

        // Broad Phase AABB test against dynamic objects
        D3DXVECTOR3 Min = pObject->BoundsMin, Max = pObject->BoundsMax;

        MathUtility::TransformAABB( Min, Max, pObject->CollisionMatrix );
```

As we now have the current dynamic object's box in world space and we also stored the ellipsoid's world space bounding box extents in the m_vecEllipsoidMin and m_vecEllipsoidMax member variables at the start of the function, we can perform the AABB/AABB intersection test and skip all the dynamic object collision code that we discussed in the previous lesson. That is, if the boxes do not intersect, we skip to the next iteration of the loop and the next dynamic object to test. The remainder of the function is identical to the prior version.

```
        if ( !AABBIntersectAABB( Min,
                                 Max,
                                 m_vecEllipsoidMin,
                                 m_vecEllipsoidMax ) ) continue;

        // Calculate our adjustment vector in world space. This is for our
        // velocity adjustment for objects so we have to work in the original
        // world space.
        vecEndPoint = (Vec3VecScale( eCenter, Radius ) +
                       Vec3VecScale( eVelocity, Radius ));

        // Transform the end point
        D3DXVec3TransformCoord( &eAdjust, &vecEndPoint, &pObject->VelocityMatrix );

        // Translate back so we have the difference
```

```
        eAdjust -= vecEndPoint;

        // Scale back into ellipsoid space
        eAdjust  = Vec3VecScale( eAdjust, InvRadius );

        // Perform the ellipsoid intersect test against this object
        ULONG StartIntersection =
              EllipsoidIntersectBuffers( *pObject->pCollVertices,
                                        *pObject->pCollTriangles,
                                         eCenter,
                                         Radius,
                                         InvRadius,
                                         eVelocity - eAdjust,
                                         eInterval,
                                         Intersections,
                                         IntersectionCount,
                                         &pObject->CollisionMatrix );

        // Loop through the intersections returned
        for ( i = StartIntersection; i < IntersectionCount; ++i )
        {
            // Move us to the correct point (including the objects velocity)
            // if we were not embedded.
            if ( Intersections[i].Interval > 0 )
            {
                // Translate back
                Intersections[i].NewCenter      += eAdjust;
                Intersections[i].IntersectPoint += eAdjust;

            } // End if not embedded

            // Store object
            Intersections[i].pObject = pObject;

        } // Next Intersection

    } // Next Dynamic Object
```

Once each dynamic object has been tested and the results added to the Intersections array, we call EllipsoidIntersectBuffers one more time to perform the narrow phase tests on any geometry stored in the collision system's static geometry buffers. As discussed, now that we will be storing our static geometry in the spatial tree, these buffers will be empty when the collision system is being used by our applications. However, the logic is still supported for applications that may wish to use the collision library without having the implementation of a spatial tree forced upon them. Every polygon in these static geometry buffers is passed straight to the narrow phase, so there will be no broad phase implementation for geometry stored here. Since we have spatial trees at our disposal now, you will probably not want to store geometry in these buffers in future applications.

```
    // Perform the ellipsoid intersect test against our static scene
    EllipsoidIntersectBuffers( m_CollVertices,
                               m_CollTriangles,
                               eCenter,
                               Radius,
                               InvRadius,
```

```
                                eVelocity,
                                eInterval,
                                Intersections,
                                IntersectionCount );
```

As discussed in the previous lesson, if the caller requested that the intersection information be returned in world space (instead of ellipsoid space), we loop through each intersection that we have collected and transform the new position, collision normal, and intersection point into world space.

```
    // If we were requested to return the values in normal space
    // then we must take the values back out of ellipsoid space here
    if ( !bReturnEllipsoidSpace )
    {
        // For each intersection found
        for ( i = 0; i < IntersectionCount; ++i )
        {
            // Transform the new center position and intersection point
            Intersections[ i ].NewCenter =
                                Vec3VecScale( Intersections[i].NewCenter,
                                              Radius );

            Intersections[ i ].IntersectPoint =
                                Vec3VecScale( Intersections[ i ].IntersectPoint,
                                              Radius );

            // Transform the normal
            D3DXVECTOR3 Normal = Vec3VecScale( Intersections[ i ].IntersectNormal,
                                               InvRadius );

            D3DXVec3Normalize( &Normal, &Normal );

            // Store the transformed normal
            Intersections[ i ].IntersectNormal = Normal;

        } // Next Intersection

    } // End if !bReturnEllipsoidSpace

    // Return hit.
    return (IntersectionCount > 0);
}
```

As you can see, the changes to this core detection step method have been minimal. We will now discuss the new methods used by this function for intersection testing with the spatial tree.

CollectTreeLeaves – CCollision

As we saw in the previous function, if a spatial tree has been registered with the collision system, the CollectTreeLeaves method is called to fill the collision system's leaf array with all the leaves which intersect the world space bounding box of the swept ellipsoid. This function is extremely small. It first makes sure that the leaf list is emptied so that no leaf pointers exist in the leaf from a previous query.

Then this leaf list is passed into the spatial tree's CollectLeavesAABB method along with the world space AABB of the swept ellipsoid.

```
void CCollision::CollectTreeLeaves( )
{
    // Clear the previous leaf list
    m_TreeLeafList.clear();

    // Collect any leaves we're intersecting
    m_pSpatialTree->CollectLeavesAABB( m_TreeLeafList,
                                       m_vecEllipsoidMin,
                                       m_vecEllipsoidMax );
}
```

When this function returns program flow back to the EllipsoidIntersectScene method, the CCollision::m_TreeLeafList member will contain a list of pointers for all leaves that were intersecting the ellipsoid's AABB.

EllipsoidIntersectTree – CCollision

When the CollectTreeLeaves method returns back to EllipsoidIntersectScene, all the leaves have been collected and stored. Next a call is made to the EllipsoidIntersectTree method which iterates through every polygon in those leaves and performs the narrow phase test. Although this function code looks quite large, for the most part it is an exact duplicate of the EllipsoidIntersectBuffers method used to perform the narrow phase test on the other geometry pools. The reason that most of that code has been cut and pasted into this function is that we have to do some additional testing and are working with slightly different source data.

The ellipsoid's eSpace center position and velocity vector are passed in along with the ellipsoid radius and inverse radius vectors. We are also passed a float reference that is used to return the *t* value of intersection back to the caller. The function also requires an array of CollIntersect structures which it will populate when intersections are found that occur at a closer *t* value that the one passed in the eInterval parameter. The final parameter contains the number intersections currently in this array. As you have probably noticed, the parameter list is identical to EllipsoidIntersectBuffers, which we discussed in detail in the previous lesson. That is because this is essentially the same function.

The first thing we do is store the current number of intersections in the passed array in the FirstIndex local variable. This is so we know where we have started adding structures so that we can return this information back to the caller. We then store the current value of the application counter in a local variable so that we can use it to make sure that polygons that exist in multiple leaves do not get tested more than once.

```
ULONG CCollision::EllipsoidIntersectTree( const D3DXVECTOR3& eCenter,
                                          const D3DXVECTOR3& Radius,
                                          const D3DXVECTOR3& InvRadius,
                                          const D3DXVECTOR3& eVelocity,
                                          float& eInterval,
```

```
                                        CollIntersect Intersections[],
                                        ULONG & IntersectionCount )
{
    D3DXVECTOR3 ePoints[3], eNormal;
    D3DXVECTOR3 eIntersectNormal, eNewCenter;
    ULONG       NewIndex, FirstIndex, CurrentCounter;
    long        i, j;
    bool        AddToList;

    // FirstIndex tracks the first item to be added the intersection list.
    FirstIndex = IntersectionCount;

    // Extract the current application counter to ensure duplicate
    // polys are not tested multiple times in the same iteration / frame
    CurrentCounter = GetGameApp()->GetAppCounter();
```

Now we set up a loop to iterate through each collected leaf in the leaf list. For each leaf, we extract its polygon count and set up a loop to iterate though every polygon in that leaf. In this inner loop, we fetch a pointer to the current CPolygon we are going to test in the current leaf we are processing.

```
    // Iterate through our triangle database
    ISpatialTree::LeafList::iterator Iterator = m_TreeLeafList.begin();
    for ( ; Iterator != m_TreeLeafList.end(); ++Iterator )
    {
        ILeaf * pLeaf = *Iterator;
        if ( !pLeaf ) continue;

        // Loop through each polygon
        for ( i = 0; i < (signed)pLeaf->GetPolygonCount(); ++i )
        {
            // Get the polygon
            CPolygon * pPoly = pLeaf->GetPolygon( i );
            if ( !pPoly ) continue;
```

Next we test the polygon's m_nAppCounter member. If we find that it is the same as the current application counter value, it means we have already processed this polygon when visiting an earlier leaf in the list. Therefore, we skip any further testing on this polygon and go to the next polygon in the list.

```
            // Skip if this poly has already been processed on this iteration
            if ( pPoly->m_nAppCounter == CurrentCounter ) continue;
            pPoly->m_nAppCounter = CurrentCounter;
```

If we make it this far in the inner loop, it means we have not yet tested the current polygon. We now perform an additional broad phase step which is an AABB/AABB intersection test between the polygon's bounding box and the bounding box of the swept ellipsoid. If the bounding boxes do not intersect then the swept ellipsoid cannot possibly intersect the polygon, so we skip to the next iteration of the loop. With large leaf sizes, this minimizes the number of polygons passed to the narrow phase test and generates a considerable performance increase.

```
            // Are we roughly intersecting the polygon?
            if ( !AABBIntersectAABB( m_vecEllipsoidMin,
                                     m_vecEllipsoidMax,
                                     pPoly->m_vecBoundsMin,
```

```
                                        pPoly->m_vecBoundsMax ) ) continue;
```

If we make it this far, then we have a polygon that may well intersect the ellipsoid. Thus it will need to be passed on to the narrow phase.

Our next step would usually be to transform the vertices of the triangle into eSpace and then pass this triangle into the SphereIntersectTriangle method. However, our tree does not store triangles; it stores clockwise winding N-gons that describe triangle fans with an arbitrary number of triangles. For example, a polygon might be a clockwise winding of four vertices describing a quad comprised of two triangle primitives. Since our SphereIntersectTriangle method only works with triangles and not N-gons (as its name certainly suggests), we will need to set up an inner loop that will iterate though every triangle in the polygon and send it to the SphereIntersectTriangle method, one at a time. For example, in the case of a quad, we would need to call SphereIntersectTriangle twice, once for its first triangle and once for its second triangle.

To calculate how many triangles comprise a clockwise winding N-gon we simply subtract 2 from the number of vertices. Figure 14.92 demonstrates this relationship by showing a clockwise winding N-gon with 8 vertices and 6 triangles.

As you can see, we can find the vertices for each triangle by stepping around the vertices in a clockwise order. The first triangle will be formed by the first three vertices, and from that point on, we can just advance the last two vertices to form the next triangle. We can see for example that the first vertex V1 forms the first vertex for all triangles. The other two vertices are found for every triangle simply by stepping around the edges. For example, the last two vertices of the first triangle are v2 and v3. The last two vertices of the second triangle are v3 and v4, and so on.

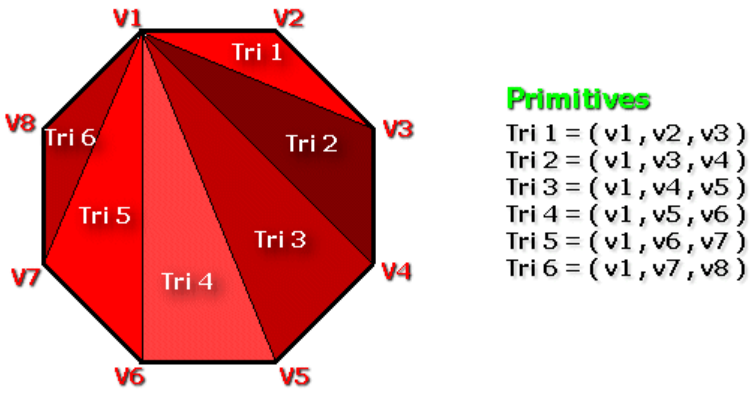


Figure 14.92

In the next section of code we start by transforming the polygon's normal into eSpace. We then set up a loop to iterate through every triangle in the polygon. In each iteration we select the three vertices that form the current triangle we are processing (see Figure 14.92) and transform them into eSpace before sending them into the SphereIntersectTriangle method.

```
            // Transform normal and normalize
            eNormal = Vec3VecScale( pPoly->m_vecNormal, Radius );
            D3DXVec3Normalize( &eNormal, &eNormal );

            // For each triangle
            for ( j = 0; j < pPoly->m_nVertexCount - 2; ++j )
            {
                // Get points and transform into ellipsoid space
                ePoints[0] = Vec3VecScale( (D3DXVECTOR3&)pPoly->m_pVertex[0],
```

```
                                    InvRadius );

            ePoints[1] = Vec3VecScale( (D3DXVECTOR3&)pPoly->m_pVertex[ j + 1 ],
                                    InvRadius );

            ePoints[2] = Vec3VecScale( (D3DXVECTOR3&)pPoly->m_pVertex[ j + 2 ],
                                    InvRadius );

            // Test for intersection with a unit sphere and the
            // ellipsoid space triangle

            if ( SphereIntersectTriangle( eCenter,
                                          1.0f,
                                          eVelocity,
                                          ePoints[0],
                                          ePoints[1],
                                          ePoints[2],
                                          eNormal,
                                          eInterval,
                                          eIntersectNormal ) )
            {
```

Note: The above code could be optimized by pre-transforming the vertices once, storing them in a local buffer, and then subsequently iterating through the triangles.

If the SphereIntersectTriangle method returns true then the sphere does intersect the triangle and eInterval will contain the *t* value of intersection. If eInterval is larger than zero then the intersection happened at some distance along the ray and we can calculate the new non-intersecting position of the sphere by adding the velocity vector scaled by the *t* value to the original centerpoint of the sphere. If eInterval is less than zero, then the sphere is embedded inside the triangle and eInterval describes the distance to move the sphere position along the direction of the plane normal in order to un-embed it.

```
                // Calculate our new sphere center at the point of intersection
                if ( eInterval > 0 )
                    eNewCenter = eCenter + (eVelocity * eInterval);
                else
                    eNewCenter = eCenter - (eIntersectNormal * eInterval);
```

The rest of the function is identical to the EllipsoidIntersectBuffers method. If there are currently no intersections stored in the intersection array, or if the *t* value for this intersection is less than the *t* value for the intersections currently stored in this array, we add the intersection information to the first element in the intersections array and set the intersection count to zero.

```
                // Where in the array should it go?
                AddToList = false;
                if ( IntersectionCount == 0 ||
                     eInterval < Intersections[0].Interval )
                {
                    // We either have nothing in the array yet, or the new
                    // intersection is closer to us
                    AddToList         = true;
                    NewIndex          = 0;
                    IntersectionCount = 1;
```

```
                    // Reset, we've cleared the list
                    FirstIndex        = 0;

                } // End if overwrite existing intersections
```

Alternatively, if eInterval is the same as the *t* value for the intersections stored in the Intersections array, then we add it to the end of the list and increase the intersection count member.

```
            else if ( fabsf( eInterval - Intersections[0].Interval )< 1e-5f )

              {
                    // It has the same interval as those in our list already,
                    // append to
                    // the end unless we've already reached our limit
                    if ( IntersectionCount < m_nMaxIntersections )
                    {
                        AddToList         = true;
                        NewIndex          = IntersectionCount;
                        IntersectionCount++;

                    } // End if we have room to store more

                } // End if the same interval
```

If we get to this point in the inner loop and find that the AddToList boolean has been set to true, it means the current triangle being tested was either closer or at the same distance along the ray as those intersections currently stored in the intersection array. In this case, we fill out the element in the array with the eSpace intersection information for the triangle.

```
                // Add to the list?
                if ( AddToList )
                {
                    Intersections[ NewIndex].Interval     = eInterval;

                    Intersections[ NewIndex].NewCenter    = eNewCenter +
                                                            (eIntersectNormal
                                                            * 1e-3f);

                    Intersections[ NewIndex].IntersectPoint = eNewCenter –
                                                             eIntersectNormal;

                    Intersections[ NewIndex].IntersectNormal= eIntersectNormal;
                    Intersections[ NewIndex].TriangleIndex   = 0;
                    Intersections[ NewIndex].pObject         = NULL;

                } // End if we are inserting in our list

            } // End if collided

        } // Next Triangle

    } // Next Polygon
```

```
    } // Next Leaf

    // Return hit.
    return FirstIndex;
}
```

We have now covered all the changes to the collision system. The end result is a powerful collision query engine that we can use in all future lab projects. The difference in query times with the spatial tree broad phase is quite significant. You can test this for yourself when you experiment with the lab project source code.

14.23 The TransformAABB Math Utility Function

In the EllipsoidIntersectScene method we saw a call being made to the MathUtility::TransformAABB method. This call transformed the AABB of a dynamic object into world space. The method was passed the minimum and maximum extents of a bounding box and a transformation matrix with which to transform the AABB.
Let us be clear about exactly what this function is supposed to do. It does not actually transform the box in the traditional sense, as an AABB by its very nature is always aligned with the world axes. If we were to rotate the box we would in fact have an OBB (oriented bounding box). The OBB would have to be stored in a very different way (position, extents, and orientation of its local axes) and this is not our goal. OBBs are much trickier to work with, and generally more expensive (although tighter fitting), and since our collision system is currently working with AABBs throughout, we will continue to do that.

What this function will do however is calculate a new AABB that contains the passed AABB post-rotation. To understand this concept, take a look at the AABB shown in Figure 14.93. In this example the box is assumed to be centered at the origin of the coordinate system with a maximum extents position vector of (10,10).

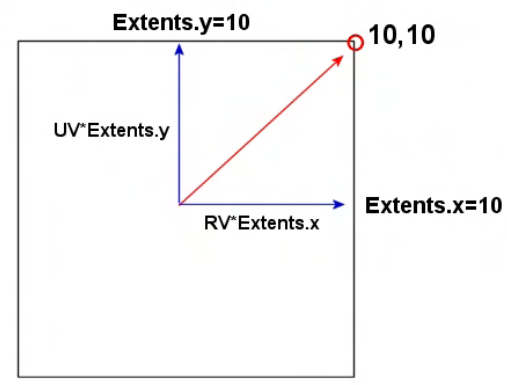


Figure 14.93

We know that an extents vector of <10,10> means the box expands to a maximum of 10 units along the world x axis and a maximum of 10 units along the world y axis. That is, we know that in the case of an AABB, Extents.x describes the position on the world X axis where the right side of the box intersects it and Extents.y describes the position along the world Y axis where the top of the box intersects it. For ease explanation we will deal with a 2D coordinate system, but the same holds true in 3D.

We know that in the case of an axis aligned bounding box, its local right and up vectors are aligned with the world X and Y axes respectively. However, if we perceive the world axes to be the local right and up vectors for the box, we can see that Extents.x describes the amount we would need to scale the box's unit length right vector (RV) to create a vector that when added to the box centerpoint describes a point

on the plane of its right face. Likewise, the unit length up vector for the box (UV), when scaled by the y component of the Extents vector and added to the centerpoint of the box, describes a point on the plane on which its top face lies. We can see that scaling the right vector and the up vector of the AABB by the components of the extents vector essentially just returns the extents vector (just as if we were transforming the extents vector by an identity matrix). However, what we want to do is transform the up and right vectors of the AABB by the passed matrix and calculate a new AABB that encompasses the OBB these new vectors describe.

In Figure 14.94 we show what the box would look like if we were to rotate these vectors 45 degrees to the left, simulating what would happen if the passed transformation matrix contained a 45 degree rotation. We can see that the rotated right and up vectors (which were scaled by the extents vectors components) still describe the location of the right and top planes of the box faces, but they are no longer aligned to the world axes. Essentially, these vectors and the center point describe an OBB now.

We know that if we rotate an AABB 45 degrees to the left or right it will pierce the original extents and a new larger AABB will need to be calculated to contain it. It is this new AABB (the red dashed box in Figure 14.93) that we want this function to efficiently calculate.

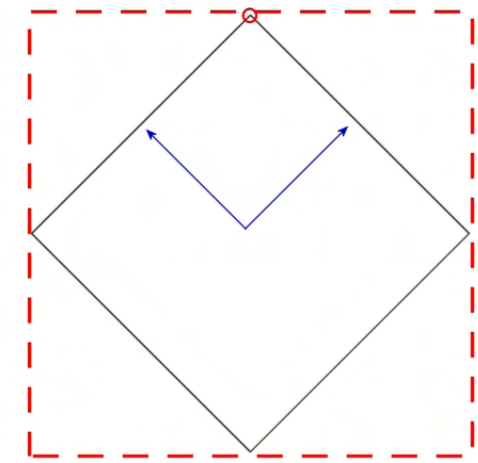


Figure 14.94

The first section of code generates the scaled and rotated right, up, and look vectors (3D) for the box.

First we calculate the center of the box and subtract it from the passed maximum extents vector. This gives us the diagonal half length of the box shown as the red arrow in Figure 14.93. We then make a copy of the passed matrix into a local variable and zero out its translation vector (we will deal with translation separately at the end of the function).

```
void MathUtility::TransformAABB( D3DXVECTOR3 & Min,
                                 D3DXVECTOR3 & Max,
                                 const D3DXMATRIX & mtxTransform )
{
    D3DXVECTOR3 BoundsCentre = (Min + Max) / 2.0f;
    D3DXVECTOR3 Extents = Max - BoundsCentre, Ex, Ey, Ez;
    D3DXMATRIX  mtx = mtxTransform;

    // Clear translation
    mtx._41 = 0; mtx._42 = 0; mtx._43 = 0;
```

Our next step is to transform the centerpoint of the box by the transformation matrix which rotates the centerpoint into the space described by the passed matrix.

```
    // Compute new centre
    D3DXVec3TransformCoord( &BoundsCentre, &BoundsCentre, &mtx );
```

Now we need to compute the scaled and rotated local axes for the box shown as the rotated blue arrows in Figure 14.94. We said that we needed to scale the unit length axes by their matching components in the extents vector so that the local up, right, and look vectors are grown in length such that they describe distances to the planes of the top, right and front faces. However, as the passed transformation matrix already contains the rotated (but unit length) right, up, and look vectors in its 1st, 2nd, and 3rd rows respectively, we can just scale the rotated unit length axes by their matching components in the extents vector to get the rotated axes:

```
    // Compute new extents values
    Ex = D3DXVECTOR3( mtx._11, mtx._12, mtx._13 ) * Extents.x;
    Ey = D3DXVECTOR3( mtx._21, mtx._22, mtx._23 ) * Extents.y;
    Ez = D3DXVECTOR3( mtx._31, mtx._32, mtx._33 ) * Extents.z;
```

Let us plug in some example values to see what has happened so far. We will once again use the 2D case for our diagrams.

Imagine we pass in a transformation matrix that will rotate our original 2D box (with a half diagonal length vector of <10, 10>) 45 degrees to the left. We know in such a case that the first row of the matrix which contains the rotated right vector will store the vector <0.707, 0.707> and the up vector stored in the matrix will be <-0.707, 0.707>. These describe the unit length vectors <1,0,0> and <0,1,0> rotated 45 degrees to the left:

RV = [0.707 , 0.707]
UV = [-0.707, 0.707]

These vectors current describe the directions we would need to travel from the box centerpoint to reach the rotated right and top faces of the box. They do not yet describe the distance over which to travel. However, in the above code we scale the up and right vectors (and the look vector in our 3D code) by their matching components in the Extents vector to generate the up and right vectors shown in Figure 14.95. These describe the direction and distance to the right and top faces of the rotated AABB (OBB).

In Figure 14.95 we see that Ex and Ey contain the results of the scaled RV and UV vectors (scaled by the original extent vector components – which are both 10 units). When added to the center position of the box, they describe points in the center of the right and top faces.

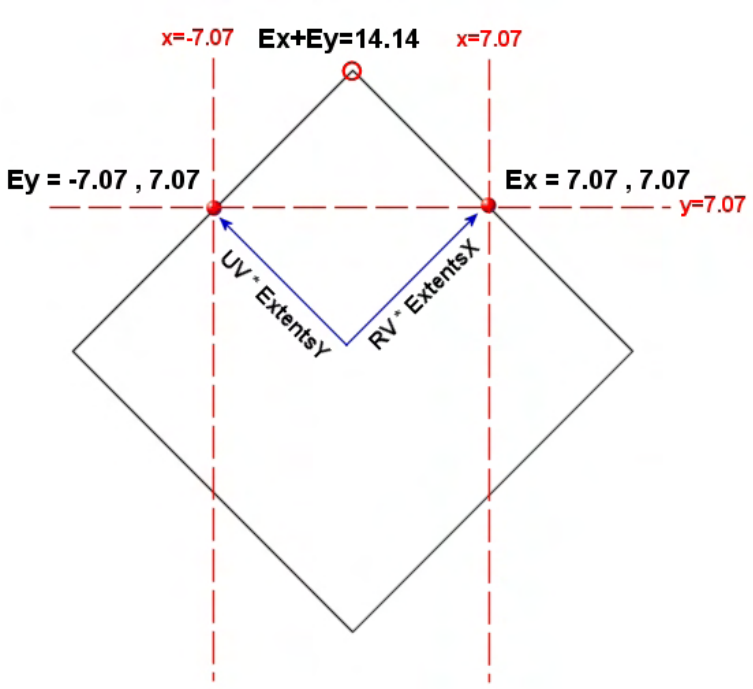


Figure 14.95

Ex = RV*10 = [7.07 , 7.07]
Ey = UV*10 = [-7.07 ,7.07]

So with these two points, how do we calculate the new extents of the rotated box along the world X and Y axes so that we can build a new AABB that encompasses it?

It just so happens that summing the absolute x components of each vector will tell us the size of the box along the world X axis and summing the y components of both vectors will tell us the height of the rotated box in world aligned space. For example:

NewExtents.x = abs(Ex.x) + abs(Ey.x) = 7.07 + 7.07 = 14.14
NewExtents.y = abs(Ex.y) + abs(Ey.y) = 7.07 + 7.07 = 14.14

As you can see, by summing the like components of the scaled and rotated vectors, we can compute exactly how much of the world X and Y axes this rotated OBB covers. NewExtents.x now describes the distance from the center of the box to the tip of the diamond on the right hand side of the rotated box. NewExtents.y describes the distance from the center of the box to the tip of the diamond at the top of the OBB (see Figure 14.96).

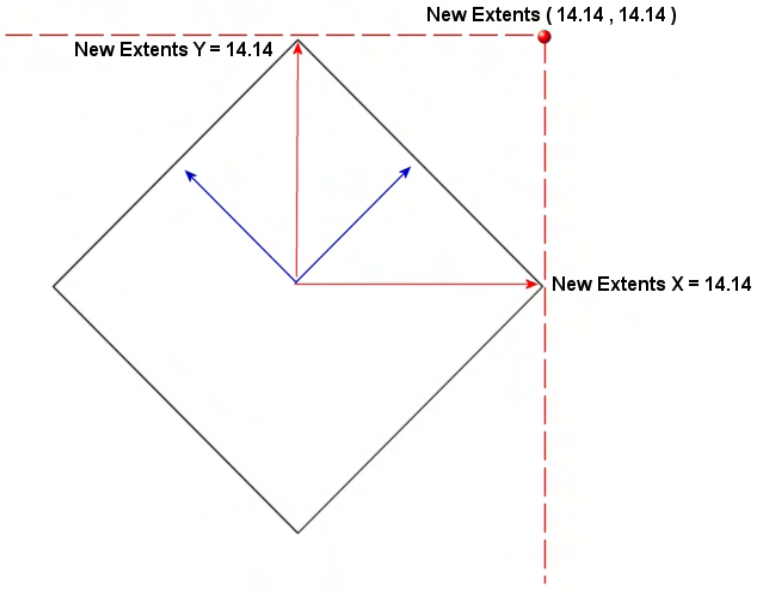


Figure 14.96

Looking at figure 14.96 we can see that the new diagonal half length of the box is therefore:

NewMaxExtents = [New Extents.x , New Extents.y] = [14.14 , 14.14]

Let us now take a look at the rest of the code to the function. In the previous section we calculated the rotated and scaled up, right, and look vectors of the OBB and stored them in the vectors Ex, Ey and Ez (the blue arrows in Figure 14.96). As we just discovered, all we have to do is sum the absolute values of the like components in each vector to get the new extents vector for the box.

```
    // Calculate new extents actual
    Extents.x = fabsf(Ex.x) + fabsf(Ey.x) + fabsf(Ez.x);
    Extents.y = fabsf(Ex.y) + fabsf(Ey.y) + fabsf(Ez.y);
    Extents.z = fabsf(Ex.z) + fabsf(Ey.z) + fabsf(Ez.z);
```

At this point, Extents holds the diagonal half length vector of the AABB we wish to create (which will bound the rotated original AABB). We can calculate the minimum extent position vector for the box by subtracting the new extents vector (diagonal half length vector) from the box center, and calculate the new maximum extents position vector by adding it to the box center. Finally, we also add the minimum and maximum extents vectors of the new AABB to the translation vector stored in the fourth row of the passed matrix. This means the new AABB is not only recalculated to encompass the rotation applied to the original AABB, but is also translated into the position described by the passed matrix.

```
    // Calculate final bounding box (add on translation)
    Min.x = (BoundsCentre.x - Extents.x) + mtxTransform._41;
    Min.y = (BoundsCentre.y - Extents.y) + mtxTransform._42;
    Min.z = (BoundsCentre.z - Extents.z) + mtxTransform._43;
    Max.x = (BoundsCentre.x + Extents.x) + mtxTransform._41;
    Max.y = (BoundsCentre.y + Extents.y) + mtxTransform._42;
    Max.z = (BoundsCentre.z + Extents.z) + mtxTransform._43;
}
```

And there we have it. This is a very handy way to calculate a new AABB for an AABB that you wish to transform into another space. The result is an AABB which completely contains the original rotated AABB. Keep in mind that this AABB has the potential to be much larger than its untransformed predecessor, which means that query accuracy can be decreased as a result of the looser fitting volume.

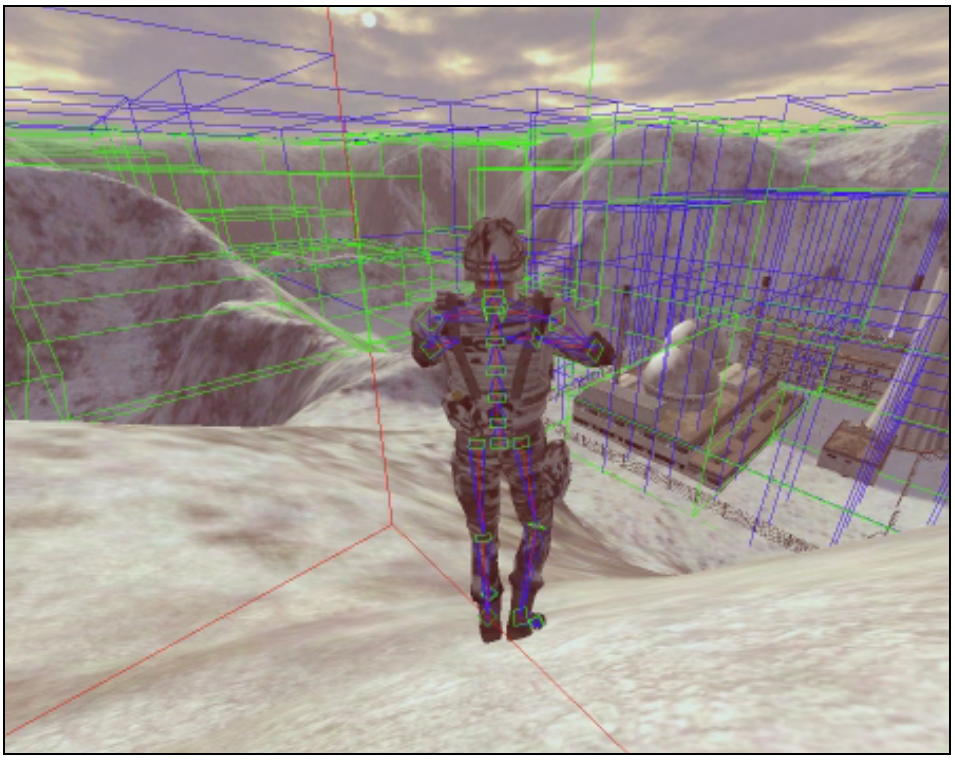
Conclusion

In Lab Project 14.1 you will see a lot of code associated with tree rendering which we have not yet covered. That is because this lab project will span two lessons, with the render system being explained in the next one. For now, you should focus your attention on the main implementation of our tree classes and the changes made to the collision system that allow us to use the spatial tree as a broad phase.

In the next lesson we will see the same spatial tree being used to speed up rendering. This will lay the foundation for a rendering system that will be utilized by our spatial trees and by the powerful BSP/PVS rendering engine we will create as we wrap up this course.

Chapter Fifteen

Hierarchical Spatial Partitioning II



Introduction

In the previous lesson we developed various spatial trees for use with our collision system, allowing us to perform polygon queries on large scenes in a fraction of the time it would have otherwise taken. In this lesson we will add to the code that we developed and implement a hardware-friendly rendering system using the same spatial trees. Since our spatial tree will ultimately be responsible for rendering its static polygon data, we will provide a means for allowing the tree to render only the polygon data that is currently visible (i.e., sections of the scene that are contained or partially contained inside the camera frustum). From the application's perspective, rendering the tree will be similar to rendering a mesh. CBaseTree will expose a DrawSubset method (just like CActor and CTriMesh) which will instruct the tree to render all visible polygons that use a given attribute in a single call. This will allow the application to set the textures and materials used by a given subset before instructing the tree to render all currently visible polygons that use it. This will ensure that the static data in our tree can be batch rendered, minimizing the texture and material state changes that have to be performed by the application.

All of the tree types we studied have the same characteristics with regards to rendering. This will make CBaseTree an ideal location for the implementation of the render system. Thus we benefit from having only one implementation of the system that is shared by all derived types. All of our trees have leaves which are either currently visible or outside the frustum. They may or may not contain polygon data which will have to be rendered if the leaf is indeed considered visible. Theoretically, our rendering system has a simple job; traversing the tree and rendering only the polygons stored in visible leaves. However as trivial as this may seem, a naïve implementation can have drastic performance implications as we will discuss in the next section. We will have to develop a system that keeps the number of traversals to a minimum and introduces as little CPU processing overhead as possible when determining which data should be rendered. On the latest graphics cards, which are capable of rendering very large numbers of triangles, we could easily end up with a situation where rendering the entire scene brute force could far outperform our hierarchical frustum culled visibility system, if an inefficient system is put in place.

In the second part of this lesson we will add dynamic object support to our spatial trees. Although the application will be responsible for rendering its dynamic objects, by storing them in the spatial tree and having access to the leaves in which their bounding volumes exist, we can benefit from the visibility system such that the application need only render a dynamic object if it currently exists in a leaf that is flagged as visible. We will add methods to CBaseTree to insert a new dynamic object into the tree which will be called when the dynamic object is first created. The tree will have no knowledge of exactly what type of object has been assigned to it; it will view the concept of dynamic objects in a rather abstract way. This will ensure that our tree will not become dependent on certain object types such as CActor or CTriMesh. From the tree's perspective, each dynamic object will simply be an AABB and a context pointer which only has meaning to the application. Methods will also be added to the tree so that the application can update the current position of a dynamic object that is registered with the spatial tree. This will instruct the tree to remove the dynamic object's pointer from any leaves in which it is currently contained and use its AABB to find the new leaves its pointer should be added to. The tree will expose methods such as GetVisibleLeaves which will return a list of only the leaves in the tree which are currently visible. The application can then loop through these leaves and render only the dynamic

objects it finds stored there. Any dynamic objects assigned to invisible leaves will not be processed at all. For each dynamic object registered with the system the tree will maintain a list containing all the leaves that object is currently contained within. This allows the application to query the leaf list for a given dynamic object so that the application has knowledge of the leaves in which it is contained (whether those leaves are visible or not). In this section, we will also cover the application code that has been changed to support dynamic objects being used with an ISpatialTree derived class.

Let us begin our discussions by tying up the spatial tree's management of its static polygon data. In the previous chapter we discussed how to add, build, and run collision queries on this data. Now we must implement the final piece, the code that will render this static geometry using hierarchical frustum culling.

15.1 Rendering Spatial Trees

When we concluded our coverage of spatial trees in the previous chapter, each leaf stored the polygons it contained as an array of CPolygon pointers. This polygon data is located in system memory to make sure that it is easily accessible for collision queries. Consequently, it is not currently able to be efficiently rendered. We know that in order to get maximum performance from our rendering system we will want the geometry stored in vertex and index buffers located in video memory so that the GPU on the graphics hardware can access that data for transformation/lighting purposes as quickly as possible. However, we certainly would not want the only copy of the data our tree uses to be stored in vertex and index buffers since this would require the locking and reading these buffers during collision queries which would result in very inefficient collision tests. What we will need to do is create a second copy of the scene data that is stored in vertex and index buffers that can be optimally rendered by the hardware. This also means that we can perform optimizations on the renderable version of the data. For example, in this lesson a key optimization will be performing a weld to get rid of duplicate vertices. This is an important step since scenes often have separate vertices for each polygon, even when polygons share an edge and have the same attributes.

The classic example of why it is efficient to perform a weld on the render geometry can be seen if you were to place a cube in GILES™ and export it to an IWF file. For various reasons, GILES™ stores all of its data at the per-face level (even vertices), which means it will never share vertices between faces. We know that if we were to place a cube and assign the same attribute to each face of that cube, we would only need to have eight vertices that are all indexed by the six faces (twelve triangles) of that cube. However, GILES™ will actually export four separate vertices per face (quad) resulting in a cube consisting of 24 vertices, regardless of vertex location, material, UV coordinates, etc. Thus, in any given corner of the cube, there exists three vertices in the same location (one for each face that meets at that corner point). For rendering purposes, it is generally going to be preferable to have a shared pool of eight vertices that are indexed by all the faces of the cube. This is essentially what a weld does for us -- it finds these duplicated vertices and replaces them with a single vertex. All the faces that used those previous vertices then have their indices remapped to the new single vertex. The cube looks exactly the same, but we have just eliminated 2/3^{rds} of our vertex data. If we imagine the savings that might be gained by welding an entire scene, we can see that this would allow us to fit much more geometry into a single vertex buffer and thus, minimize the number of vertex buffers we will need to store our scene

geometry. Remember that changing between vertex buffers can be an expensive operation, so we will want to do it as infrequently as possible. By minimizing the number of vertex buffers needed to store the scene, we minimize the number of vertex buffers we need to set and render from in a given frame.

There are many ways a rendering system can be implemented for a spatial tree. Indeed, quite a few designs were attempted during the development of this course. This was time consuming but fortunate as we discovered quite significant performance differences between the various systems. Even system designs that seemed quite clever and robust on paper performed terribly when compared against simple brute force rendering on high end machines with the latest graphics hardware. To be fair, a number of the maps and levels we tested had fairly low polygon counts (by today's standards), so we were not too surprised to see that they could be easily brute force rendered by the latest nVidia and ATI offerings. Also note that these scenes were not being rendered with complex shaders or multiple render passes, so the latest hardware could easily fill the frame buffer without any trouble. This was certainly true in tests where we used pre-lit scenes with a single texture.

Although this might seem an unfair test for any visibility system, the fact that brute force rendering was significantly outperforming these various system implementations on high end hardware, even when only half the scene was visible, is an important point that is really worth keeping in mind. It is a common mistake to assume that because the spatial visibility system might be rendering only half the scene when the player is in a given location within the game world, versus the brute force approach which is always rendering the entire scene, that the latter would always be slower. But it is critical to factor in the hardware in question. If the hardware is able to brute force render the entire scene without difficulty, the difference then becomes one of what is involved in the spatial system's determination of which polygons should be rendered. That is, we found in various spatial tree visibility systems that we implemented that the processing and collection time for visible data often exceeded the time it took to render the whole scene using brute force.

> **Note:** To be clear, we are talking solely about the rendering of the static scene polygon data here, which is perhaps being a bit unbalanced with respect to such spatial visibility schemes. One must also factor in the savings of quickly identifying and rendering dynamic objects which are only contained in visible leaves. A typical scenario might contain dozens of skinned characters for example, of which only two are contained in visible leaves. If these characters were each comprised of thousands of triangles, the savings over brute force would generally become more apparent. In this section though, we are focusing on trying to efficiently render the static scene in a way that helps rather than hinders performance on high end systems.

When the same systems were compared on much lower end machines, the situation was much better for hierarchies. That is, the extra CPU processing involved in collecting only the visible data made all the difference to the aging video hardware. Such cards could not render the entire scene brute force at interactive frame rates and the various visibility systems helped to achieve much higher frame rates. One might be tempted to conclude that the spatial tree's visibility system could be put in place to help our games run at acceptable speeds on lower end machines even if they provide no real benefit on the latest cutting edge systems (which seemed not to need much help). This is certainly a valid consideration, as it allows a game to run on a much wider variety of machines (which could mean the difference between decent sales and great sales for a commercial title). Certainly it is generally in a software company's best interest to implement systems that support a wide variety of target platforms as it opens up the potential market for the game. A recent survey showed that the vast majority of computer owners that play games

are still using nVidia geForce™ 2/3 (or equivalent) hardware, so there is certainly something to be said about that.

However, what was unacceptable about many of the system designs we tried was that while speeding up performance on low end machines, performance on high end machines was degraded (sometimes significantly so). Essentially, the time required to traverse the tree and collect the visible data became the main bottleneck. As mentioned, this situation would likely be quite different if we were rendering a huge scene with high polygon counts or a scene that required many rendering passes per–polygon to achieve advanced lighting and texturing effects. In such cases, even the latest graphics cards would find themselves hard pressed to perform brute force renders. This is where our spatial tree's visibility system could really make a difference as it only asks the hardware to render the geometry that is currently visible. This would once again allow for a reduction in the list of primitives to be rendered each frame which will hopefully represent an amount that can be easily rendered by the hardware.

We should also not neglect the far plane case when imagining how our spatial tree's visibility system can help out even powerful graphics cards. Imagine for example a game that implements a continuous terrain such as those found in such products as Microsoft Flight Simulator®. The terrains in such games would not only be too large to render brute force, but even on the latest hardware they might also be too large to even fit in memory at once. The terrain data would have to be streamed from disk as and when it comes into view and released when it is no longer visible. With our spatial tree's visibility system, any leaves that are outside the frustum could have their geometry data flushed from memory. With our hierarchical traversals, we could very easily detect which leaves are within the frustum each frame and stream the terrain data for that leaf from disk if it is not already loaded. We can then collect the data from each visible leaf and pass it to the pipeline to render. This task would be almost impossible to achieve without a spatial manager. It is for this reason that a spatial manager and visibility systems generally will be at the heart of nearly every commercial quality game title.

Although the benefits of a spatial tree for rendering purposes are numerous, there will be times when you will be rendering small and simple scenes which modern hardware can brute force render more efficiently than our spatial tree's visibility system. Although not using a spatial manager and visibility system in the days of the software engine would have been unthinkable, the latest graphics cards do their jobs very well and require less help from the application than in days gone by. As mentioned, when rendering simple scenes, trying to help the card instead of letting it just render everything can actually hinder performance in many cases. The visibility system we choose to implement then has to be one that prepares the visible data so efficiently that the time requirements are negligible in such situations. A classic example of why this is so important can be understood when we imagine the player located in the bottom left corner of the scene looking towards the top right corner of the scene with a wide FOV and a near infinite far plane. In such cases, the entire scene would most likely be visible, so the quickest solution would always be to perform an optimized brute force render of the entire scene, batched by subset. However, our visibility system would have to perform a tree traversal and collect the data at each leaf just to determine that the entire scene should ultimately be rendered anyway, just as in the brute force case. The issue is that our visibility system had to run some form of logic to ascertain this fact before it could start rendering anything. The brute force approach had no CPU processing to perform and could start rendering straight away. In such situations, the brute force approach is generally going to beat a basic frustum visibility system hands down. What we need to do is make sure that we implement a system that minimizes the damage to frame rate in these situations, but that when the player is

positioned at a location within the scene such that only a few leaves are visible (the general case), the visibility system will outperform the brute force system.

To sum up, we are going to try to design a system that significantly speeds up our rendering on both low and high end systems in the best case; in the worst case, where the scene could easily be brute force rendered by a high end graphics card, we want our implementation to speed up rendering on a low end machine but not under-perform brute force rendering on a high end system by any significant degree.

15.1.1 Different Rendering Schemes

There are many different schemes ranging from easy to difficult that can be employed to collect and render the visible data from a spatial tree. Unfortunately, the performance of such algorithms tends to vary from poor to good in that same order. In this section we will discuss some of the approaches that one might consider employing. We will discuss techniques that span the naïve to the complex and end with a discussion of the system that we ultimately ended up going implementing in CBaseTree.

The UP Approach

The simplest way to render the visible the data is to draw the system memory polygon data stored at each visible leaf. DirectX has the IDirect3DDevice9::DrawPrimitiveUP method that allows us to render data directly using a system memory user pointer. With such an approach we do not need to make a copy of the polygon data in vertex or index buffers; we can pass a pointer to each CPolygon's vertex array directly into the DrawPrimitiveUP method. One approach would be to have an empty vertex array allocated for each subset used by the tree's static data prior to performing a visibility pass. These vertex arrays would be populated with the vertex data from each CPolygon from every currently visible leaf when the tree is traversed. After the visibility traversal is complete, this collection of vertices could be rendered. This array would then be flushed before the next visibility pass is performed.

As an example, if we imagine the static data in the tree contains polygons that use five different attributes in total, our tree would use five vertex arrays as bins in which to collect the vertex data from the visible leaves and add them to the array that matches their attribute ID. These arrays/bins would be emptied prior to the visibility process being performed each frame. We could implement a ProcessVisibility method that would be called by the application prior to rendering each subset of the tree. This method would traverse the tree, testing each node's bounding box against the camera's frustum so that any node that is currently outside the frustum will not have its children traversed into. Any child nodes and leaves underneath that node in the tree will not be traversed or processed and as such, any polygon data contained down that branch of the tree will be quickly and efficiently rejected from the collection process.

When a leaf is found that is visible, its visible Boolean will be set to true and a loop through each polygon stored there will be performed. For each polygon, we will check its attribute and copy its vertex data into the matching vertex bin. So if the current polygon we are processing in a leaf had an attribute

ID of 4, we would copy its vertices (one triangle at a time) into the 5th vertex array (VertexArray[4]). After the tree has been traversed, we would have built five vertex arrays and flagged each visible leaf. At this point, the ProcessVisibility method would return program flow back to the application.

The next stage for the application would be to loop through each subset and call the tree's DrawSubset method after setting the correct device states for that attribute. For example, the application would set the texture and material for subset 0 and call the tree's DrawSubset method, passing in this attribute ID. As the vertex array has already been compiled for all visible triangles that use this subset, this method can just call the IDirect3DDevice9::DrawPrimitiveUP method passing in VertexArray[0] and the number of triangles stored in this array. We would also state that we would like it rendered as a triangle list. Once the application had called DrawSubset for each attribute used by the tree, all visible triangles would have been rendered. By collecting the visible triangles into arrays first, we minimize the number of draw calls which would otherwise be in the tens of thousands if we rendered each polygon's vertices as and when it was located in a visible leaf during the visibility pass.

So we can see that using this approach, it is the ProcessVisibility method that actually traverses the tree and collects the triangle data for each subset. The DrawSubset method would simply render the vertex bin/array that matches the passed attribute that was compiled during the visibility pass.

What is wrong with this approach?

Although this method sounds delightfully easy to implement, it is a definite example of the wrong way to design this system. Performance would be simply terrible for a variety of reasons. First, as we traverse the tree, we have to perform a system memory copy of the vertex data from each CPolygon into the appropriate vertex bin. This would need to be done for every polygon currently visible. As vertex structures can often be pretty large, these memory copies really drain performance. When the entire scene is visible, we will be copying hundreds or thousands of vertices each frame and seriously tying up the CPU. The brute force approach in such situations would have none of this overhead and could simply just render the scene prepared in static vertex and index buffers with just five draw calls.

The second major flaw with this system is the use of the DrawPrimitiveUP method. Although it looks like the use of this function handily avoids the need to employ vertex buffers, the real concern is what happens behind the scenes. When we call the DrawPrimitiveUP method, a temporary vertex buffer will be created and locked and the vertex data we pass will then have to be copied into this vertex buffer (yet more copying of memory) before it is unlocked. After unlocking this vertex buffer, the pipeline may then have to commit this data to video memory, involving yet another copy of the vertex data over the bus into video memory. Transforming all the vertex data for the currently visible scene could mean a huge number of vertices will have to be passed over the bus each time.

As you can see, although this system it simple to implement, the performance would be quite poor. With an outrageous amount of memory copying to perform, this system is typically outperformed by brute forcing even on low end hardware. Memory copying is performance killer and in this case we are implementing three major copy stages. This is clearly not the way to go.

The Dynamic Index Buffer Approach

The dynamic index buffer method is discussed in many published texts and as such is worth discussing here. With this approach we would invoke a process just after the tree is built which prepares the render data. It would allocate a static vertex buffer for maximum performance which will typically be allocated in video memory on T&L capable cards. We will also allocate a *dynamic* index buffer for each attribute used by the static scene stored in the tree. These index buffers will be flushed prior to each ProcessVisibility call.

The additional build process that is employed only once just after the tree is first built would loop through each leaf in the tree and copy the vertex data for each polygon into the static vertex buffer. As all the vertices of all polygons in all leaves will be stored in this vertex buffer we will also need to store the indices of each polygon in the leaves themselves. This is so we know for a given visible polygon, which section of the vertex buffer it is contained in and the sections that should be rendered or skipped. As each polygon is added to the vertex buffer, we will store the indices of its vertices in the leaf. After the post-build process is performed, we would have a single vertex buffer containing all the vertices of the static scene, and in each leaf we would have the indices for all the vertices of each polygon so that we know where they exist in the vertex buffer. We will also have an array of empty dynamic index buffers (one for each subset) which will be filled with the visible polygon indices during the ProcessVisibility call.

When the ProcessVisibility call is made by the application prior to rendering the tree's subsets, this function will first make sure that all the dynamic index buffers are empty. Then it would traverse the tree searching for visible leaves as in the previously described method. For each visible leaf we find, we would loop through each of its polygons and add its indices to the dynamic index buffer that is mapped to the attribute for that polygon. For example, if a given polygon had an attribute ID of 4, we would copy its indices into the fifth index buffer (indexbuffer[4]). When the ProcessVisibility method returns program flow back to the application, the tree would have the indices of all the visible triangles for a given attribute in each index buffer. The application would then bind the appropriate texture and material, set the appropriate states, and then call the tree's DrawSubset method to draw the required polygons. This method would simply call the IDirect3DDevice9::DrawIndexedPrimitive method for each dynamic index buffer it contains. We would pass DrawIndexedPrimitive the vertex buffer used by the entire scene and the index buffer of the current subset we are rendering. That index buffer would describe the indices of all the triangles in the vertex buffer that are visible for the current subset being rendered. After this has happened for each subset, all visible triangles will have been rendered. Thus, the number of DrawIndexedPrimitive calls would be equal to the number of subsets used by the polygons in the tree.

What is wrong with this approach?

This design does perform much better than the previously discussed approach due to the fact that the vertex data was already contained in a video memory static vertex buffer. This meant that the sizable vertex structures did not have to undergo the multitude of copy phases. The vertex data was created just once at application startup and committed to video memory. However, on simple levels, brute force rendering significantly outperformed this technique on high-end hardware even when half the scene was being rejected in the visibility pass. This was especially true in the case of a clipped tree where the

polygon count was raised between 30% and 80% due to the polygon splitting performed in the build process. We can see for example that in the extreme case where there was an 80% increase in the polygon count, even when half the scene was being rejected, the polygon count had almost doubled and as such, half of our visible scene contained nearly as many polygons as brute force rendering the original polygon data. Therefore, brute force was still able to render the original scene faster than our clipped spatial tree could render only half of its split scene.

When using a non-clipped tree, things were better, but we still found that the memory copies performed at each visible leaf (polygon indices copied into the index buffer) are a bottleneck. Although index data is much smaller than vertex data and not nearly as expensive to copy or commit to video memory, when the entire scene was within the frustum, we were still dealing with tens (even hundreds) of thousands of indices to copy into the index buffer during each visibility determination pass. Combined with the fact that a dynamic index buffer typically performs a little slower than a static index buffer and we still had a system that, while much improved over the previous method, suffered poor performance compared to brute force when there was medium to high visibility in non-trivial, but not overly complex, scenes.

What was clear was that in order to get better performance we needed to eliminate the copying of vertex and index data and instead store the renderable data in static vertex and index buffers in the post build phase. This would allow our vertex and index data to reside in optimal memory using an optimal format for the driver. In the next section, we will discuss the rendering system we finally wound up implementing, which provided much better performance than the previous systems.

15.2 The CBaseTree Rendering System

The CBaseTree rendering system is not as complicated as it may first seem when looking at the code. Unfortunately, the code is made significantly less trivial by the fact that multiple vertex buffers may have to be used if the vertex count of the static scene exceeds the maximum vertex buffer size supported by the hardware. This necessitates the introduction of an additional level of structures that makes things a little difficult to follow at first. Because this reduces clarity of what is essentially a rather simple system, we will first discuss the system with this complication removed. That is, we will discuss this system assuming that we can use an infinitely large vertex buffer and as such, all the static geometry in the tree can always be contained within it. Once we understand the system, we will discuss how it needs to be adapted to cope with multiple vertex buffers.

The system operates in three stages much like the previous method we discussed. The first is executed only once when the tree is first built. It is this component that copies the vertex data of every leaf into a tree owned vertex buffer and the indices of every polygon into static index buffers (one for each subset). You will recall from our previous lesson that prior to the Build function of a derived class returning, it makes a call to the CBaseTree::PostBuild method to calculate the bounding boxes for CPolygons in the tree. This method also issues a call to a new CBaseTree member called BuildRenderData as shown below.

```
bool CBaseTree::PostBuild(  )
{
    // Calculate the polygon bounding boxes
    CalculatePolyBounds( );

    // Build the render data
    return BuildRenderData( );
}
```

It is the CBaseTree::BuildRenderData method that will build the static vertex buffer containing all the vertices of the tree's static geometry. It will also create an index buffer for each subset used by the tree and store the indices of each triangle in these buffers in leaf order. Let us discuss how our system will work and what processing this initial stage will have to perform.

This function will create a static vertex buffer to contain the vertex data and N static index buffers, where N is the number of attributes used by the static tree geometry. It will then loop through each leaf in the tree. For each leaf it is currently processing it will loop through all the CPolygon structures stored there. For each polygon, it will add its vertices to the vertex buffer and will test the attribute ID of the polygon so that its indices are generated and added to the correct index buffer. For example, if we are processing a polygon that uses attribute 4, we will add its vertices to the vertex buffer and then generate the indices for each triangle in the polygon based on the position it has been placed within the vertex buffer. Once we have the indices generated for each of its triangles we will add them to the fifth index buffer (index buffer 4) since this is the index buffer that will contain the indices of all triangles that use attribute 4.

The next bit is vitally important. When we add the indices of each polygon in the leaf to their respective index buffers, we will store (in the leaf) the first index where the triangles for this leaf start in each index buffer it uses. For example, if we are processing a leaf and we find that there are 6 triangles that use subset 2, we will add the indices of those six triangles to index buffer [1] and the leaf will be given the index where its indices begin in that index buffer and the number of primitives it contains for that leaf. As we process each leaf, we know for example that for a given subset that has polygons in leaf A, all of leaf A's indices will be added to that subset's associated index buffer in a continuous block. Going back to our previous example, if we find that a leaf has 6 triangles that use attribute 2 and the index buffer already contains 100 indices that were added when processing other leaves, the leaf will keep track of the fact that for subset 2, its polygons start at index 100 and the next 6 triangles from that point in this index buffer are its primitives. We will see why this is important in a moment.

Therefore, after this process is complete, if a given leaf contains polygons that use 4 subsets for example, that leaf will store an array of four RenderData structures (one for each subset it contains). Each RenderData structure contains the index at which this leaf's indices begin for a given subset in the associated index buffer and the number of triangles in that subset contained in the leaf. Thus, each RenderData structure stored in the leaf describes an index buffer range for a given subset. If the RenderData item stored in a leaf has an attribute ID of 5, it will describe a block of indices in the 6th index buffer (the static index buffer for triangles of subset 5) where this leaf's triangles start and end. We know then that if this leaf is visible, this section of that index buffer will need to be rendered.

We will also introduce a new class called CLeafBin. CLeafBin is really a wrapper class around an index buffer for a given subset containing methods to render its index buffer efficiently. As we need to run the collection of our render data as fast as possible, each RenderData item stored in a leaf will also store a pointer to the leaf bin for which it is associated. This way, during the traversal, when a visible leaf is found, for each RenderData structure stored (one for each subset contained in that leaf) in the leaf we can access the pointer to the associated leaf bin and inform it of a range in its index buffer that will need to be rendered.

Figure 15.1 shows a simplified diagram of the rendering system. In this example we will assume the tree contains only four leaves and uses only three attributes (the texture images describe the attributes). Study the image and then we will discuss its components.

In this example the tree contains only four leaves and these leaves can be seen in the top left corner of the image. Leaves 1 through 3 contain polygon data from two different subsets and leaf 4 contains polygon data from only one. The line dividing leaves 1 through 3 down their center is being used to indicate that these leaves have two RenderData structures because these leaves contain polygons from two different subsets. The fourth leaf contains polygon data from only the second subset and as such contains only one RenderData structure. At the bottom of the image we can see that as the tree's polygon data uses only three subsets, we allocate three leaf bins. Each leaf bin contains the static index buffer that will contain the indices of all the triangles in the tree that use that subset. Just like the tree's vertex buffer, these index buffers will also be populated just once after the tree has first been built. Do not worry about what the RenderBatch structures in each leaf bin are for at the moment; we will see how these will be used a little later to collect runs of visible triangles during the visibility determination process.

When the tree is first compiled, the leaf bin index buffers and the tree's vertex buffer will be empty. At this point, no leaves will contain RenderData structures. It is the job of the CBaseTree::BuildRenderData function to allocate these structures and populate the index buffers of each leaf bin and the vertex buffer of the spatial tree with data.

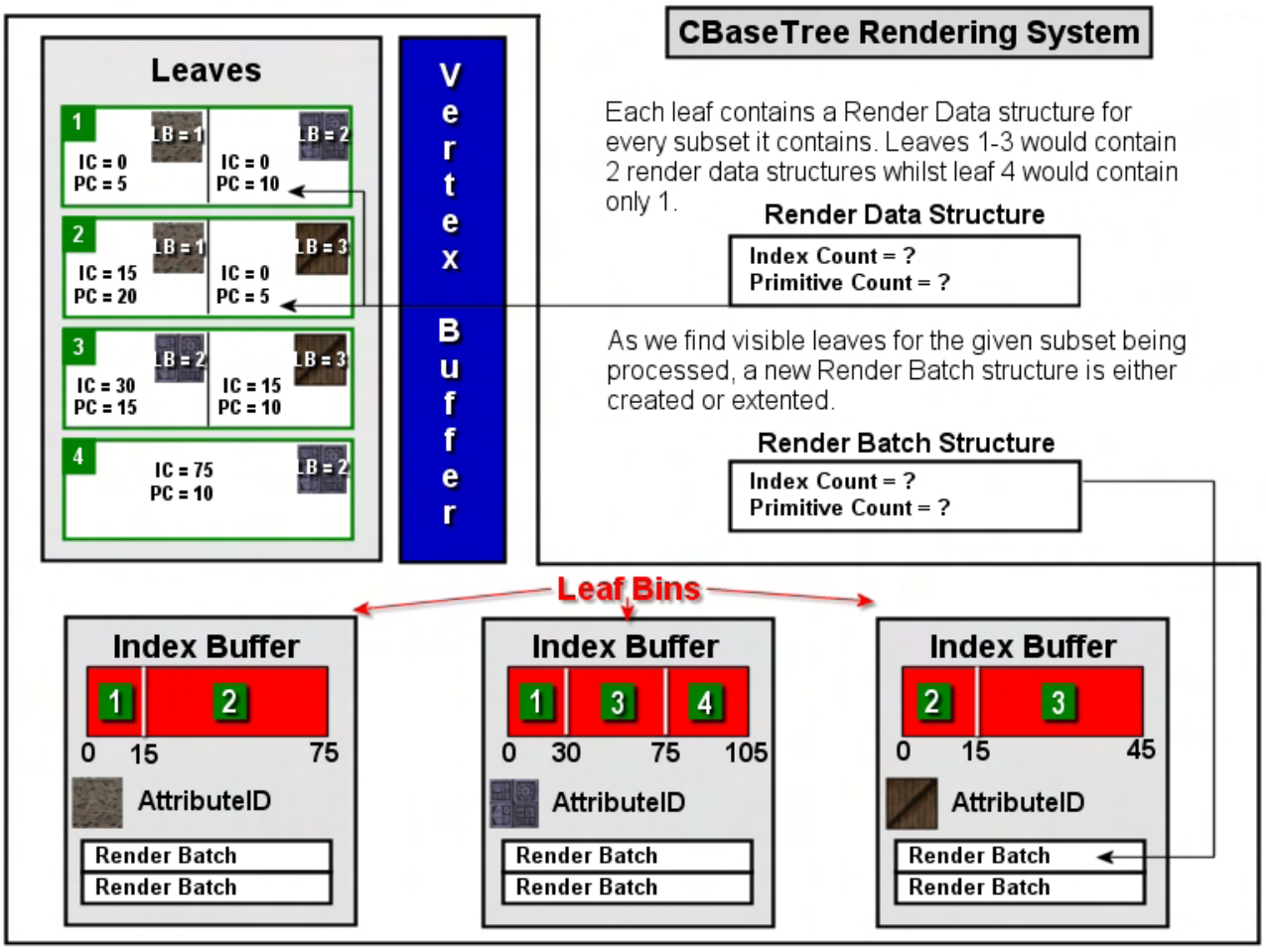


Figure 15.1

When the BuildRenderData function is first called, let us assume that it processes leaf 1 first. As it loops through each polygon, it finds that it contains 5 triangles that use subset 1 (the brick texture). It knows that it has to add the vertices of each of these triangles to the vertex buffer and the indices of each triangle (15 indices in total) to the index buffer stored in leaf bin 1 (the leaf bin associated with the first subset). It then notices as it adds these 15 indices to the index buffer that there are currently no indices in this index buffer. This means that this leaf's indices for subset 1 will be stored at the beginning of the index buffer contained in leaf bin 1. A new RenderData structure is allocated and added to the leaf which contains the subset ID of the leaf bin this data has been added to, the index into the index buffer of this leaf bin where the leaf's indices begin, and the number of triangles it has added to this index buffer. We can see if we look at the left half of leaf 1 in the diagram that its RenderData structure for leaf bin 1 contains an index count of 0, which describes where this leaf's subset 1 polygons begin in leaf bin 1 index buffer. It also contains a primitive count of 5. If we look at leaf bin 1, we can see that all 15 indices for the 5 triangles in leaf 1 have been stored at the beginning leaf 1's index buffer, exactly as described by the leaf's RenderData structure for this leaf bin.

Next we find that leaf 1 also contains 10 triangles which use subset 2. The vertices of each of these triangles are added to the vertex buffer and the indices of each vertex in the leaf that uses this subset are

generated and added to the index buffer of leaf bin 2. We are assuming in this example that these polygons are all triangles and as such this would add 30 indices to the index buffer in leaf bin 2. Once again, we create a new RenderData structure and store in it the leaf. This second RenderData structure will contain the leaf's information for leaf bin 2. We also notice that as this is the first time this subset has been encountered, there are currently no indices in the index buffer of leaf bin 2, so the RenderData structure for this leaf bin will store an index count of 0 and a primitive count of 10. At this point, leaf 1 has been processed. We know that it contains a RenderData structure for each subset it uses and each of these structures describes where its triangles exist in the associated leaf bin's index buffer. We can see for example that if this leaf is visible, when the application calls ISpatialTree::DrawSubset and passes in a subset ID of 1, we must make sure that the 5 triangles described by the first 15 indices in leaf bin 1's index buffer are rendered since these contain the triangles in leaf 1 for this subset. Likewise, when the application calls the DrawSubset method again to ask the tree to render subset 2, we know that if leaf 1 is visible, we must render the first 30 indices in leaf bin 2's index buffer.

We then move on to leaf two where we find 20 triangles that use subset 1 and 5 triangles that use subset 3. We add the vertices of each triangle to the vertex buffer and generate the indices for each triangle so that they correctly index into that vertex buffer. The 60 indices generated for the 20 triangles that belong to subset 1 are added to leaf bin 1's index buffer and a new RenderData structure is created and linked to this leaf bin via an attribute ID. Because there are already 15 indices in this index buffer, we know that the 60 indices for leaf 2's triangles must start at index 15 in leaf bin 1's index buffer. Therefore, we store an index count of 15 and a primitive count of 20 in this leaf's RenderData structure. This structure now tells us that the indices for the subset 1 triangles stored in this leaf begin at position 15 in leaf bin 1's index buffer and that there are 20 of them. We then process the other polygons in leaf 2 which in this example use subset 3. As this leaf bin has not yet had any data added to its index buffer, we know that it was empty before we added our indices for this leaf and therefore, a new RenderData structure is allocated and linked to this leaf bin via the subset ID. The index count of this structure is set to 0 as this leaf's indices for subset 3 polygons start at the beginning of this index buffer, and the primitive count is set to 5. The two RenderData structures stored in leaf 2 tell us that it contains polygons from two subsets, subsets 1 and 3. The indices of the triangles in this leaf that use subset 1 begin at index 15 in leaf bin 1's index buffer and there are 20 of them. We know that we can render this block of triangles if this leaf is visible. The second RenderData structure in leaf 2 tells us that this leaf also contains 5 triangles from subset 3 that start at position 0 in leaf bin 3's index buffer.

After this process has been repeated for all polygons in all leaves we will have the following:

- A static vertex buffer containing the vertices of all triangles in the tree.
- A Leaf Bin for every subset used by the tree's static data. Each leaf bin contains a static index buffer containing the indices of the tree's polygon data assigned to this subset.
- Each leaf will contain a RenderData structure for every subset its polygons use.
- Each RenderData structure stored in a leaf will describe where that leaf's indices start in its associated leaf bin and the number of triangles stored there (starting at that index).

If you refer back to Figure 15.1 you should be able to see how the index counts are calculated for each RenderData structure in each leaf. We can see for example leaf 4 contains only triangles that use subset 3. The indices of these triangles start at position 75 in the index buffer of leaf bin 2 and there are 10 triangles * 3 indices = 30 of them. The reason this leaf's indices start at 75 in leaf bin 2's index buffer is

that leaf 1 added 5*3=30 indices to this buffer first, followed by leaf 3 which added a further 15*3=45, taking the number of indices stored in this buffer up to 75 prior to leaf 4 being processed.

If you examine the contents of the three leaf bins in Figure 15.1 this should further clarify the relationship. At the end of the render data building process in this example, leaf bin 1 would contain 75 indices. The first 15 would describe the 5 triangles in leaf 1 which should be rendered if leaf 1 is deemed visible during the visibility processing pass. The indices from 15 to 75 describe the 20 triangles added to this index buffer from leaf 2. Therefore, we know that if leaf 2 is visible, indices 15 through 75 in leaf bin 1 should be rendered (when DrawSubset is called and passed a subset ID of 1). Furthermore, we can see that the RenderData structures in leaves 1 and 2 linked to this leaf bin describe these index ranges.

If we look at leaf bin 2 we can see that its index buffer contains triangles from three leaves and therefore, three leaves will contain RenderData structures linked to this leaf bin. Leaf 1 has added 30 indices to the beginning of this leaf bin to describe its 10 triangles that use subset 2. Leaf 3 also added 45 indices to this index buffer (starting at position 30) which describes its 15 triangles that use this subset. Finally, we can see that leaf 4 added a further 30 indices to this leaf bin describing the 10 triangles that it contains that belong to subset 2. Before continuing, study the diagram and make sure you understand the relationship between the leaf bins and the RenderData structures stored in each leaf.

The process described above is all performed by the CBaseTree::BuildRenderData method. This function is called once the tree is compiled and it builds and populates the RenderData structures stored at each leaf, the leaf bins for each subset, and the global vertex buffer that the index buffers in all leaf bins index into. The logical next question is, with this static data at our disposal, how do we efficiently collect and process it during the ISpatialTree::ProcessVisiblity call in a way that it can be efficiently rendered during a call to the ISpatialTree::DrawSubset method?

15.2.1 Render Batches

The heart of the rendering system lies in the leaf bin's ability to dynamically generate and store RenderBatch structures every time a visibility pass is performed on the tree (via a call to the ProcessVisibility function). A render batch essentially describes a list of triangles in a given leaf bin which are visible and stored consecutively in its index buffer. We can see in Figure 15.1 that render batch structures are generated and stored in the leaf bins (this takes place during the visibility pass). Each RenderBatch structure will describe a single call to the IDirect3DDevice9::DrawIndexedPrimitive method. Let us just take a look at that method to refresh our memory about its parameters. We will then be able to more easily see why a single RenderBatch structure represents a single call to this function.

```
HRESULT DrawIndexedPrimitive
(
    D3DPRIMITIVETYPE Type,
    INT BaseVertexIndex,
    UINT MinIndex,
    UINT NumVertices,
    UINT StartIndex,
    UINT PrimitiveCount
);
```

The important parameters to focus on, at least with respect to the current topic being discussed, are the last two. We pass in the location of the first index of a block of triangles we wish to render as the StartIndex parameter. The last parameter describes how many triangles (starting at StartIndex) we would like to render. Therefore, every time we call this method to render a series of triangles, the indices of those triangles must be consecutively ordered in the index buffer. For example, if we have an index buffer and we wish to render triangles 0 to 10 and triangles 40 to 50, we will have to issue two calls to this method, as shown below.

```
pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                               0,
                               0,
                               NumOfVertices,
                               0,
                               10 );

pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                               0,
                               0,
                               NumOfVertices,
                               120,            // 40*3
                               10 );
```

In the above example, NumOfVertices is assumed to contain the number of vertices in the vertex buffer bound to the device for these calls.

What is apparent by looking at the above example is that because our vertex and index buffers are static, when collecting the indices that need to be rendered from each visible leaf, we will not be able to render the triangles from multiple visible leaves with a single draw call unless these indices have been added to the index buffers of each leaf bin such that their indices follow on consecutively in the index buffer. Therefore the job of the leaf bins during a visibility pass will be to allocate RenderBatch structures to try to maximize the number of primitives that can be rendered in a single batch by detecting adjacent runs of visible triangles in the index buffer from different leaves. This will minimize the number of draw calls that will need to be made.

A RenderBatch structure is a very simple structure at its core that simply stores a start index and a primitive count as shown below. Essentially, each RenderBatch structure just represents a block of adjacent triangles in the index buffer that are currently considered visible and can be rendered via a single draw call.

```
struct RenderBatch
{
    ulong IndexCount;
    ulong PrimitiveCount;
};
```

To understand how it works, let us go back to the example tree illustrated in Figure 15.1 and assume for now that all four leaves are visible and that all the render data has been constructed as previously described. When the ProcessVisibility function is called by the application to determine leaf visibility,

the first thing it does is clear each leaf bin's RenderBatch list. That is because the job of the visibility pass will be to construct a list of render batch structures which can be executed to render the visible data.

The ProcessVisibility method will traverse the hierarchy searching for visible leaves. When a leaf is encountered which is visible, it will loop through each RenderData structure stored in that leaf. Remember, the number of RenderData structures stored in the leaf will be equal to the number of attributes/subsets used by the polygons in that leaf. Each RenderData structure would also store a pointer to the leaf bin to which it is assigned.

In the above example we would visit leaf 1 first and would determine that it is visible. We would then loop through the number of RenderData structures stored there, which in this example would be two. The first RenderData structure describes the index start and primitive count of all the triangles in leaf 1 that are stored in leaf bin 1. Therefore, we would call the leaf bin's AddVisibleData method, passing it the index count of the RenderData item and the number of primitives (shown in the following pseudo code). That is, we are informing the leaf bin of a run of visible triangles in its index buffer.

```
if (pLeaf->IsVisible)
{
   // Loop through each Render Data structure in this leaf
   for ( i = 0; i < pLeaf->RenderDataCount; i++  )
   {
       // Get current Render Data structure
       pRenderData = pLeaf->RenderData[i];

       // Fetch the leaf bin this Render Data structure is linked to
       pLeafBin = pRenderData.LeafBin;

       // Pass the index start location and primitive count into leaf bin
       pLeafBin->AddVisibleData( pRenderData->IndexCount ,
                                 pRenderData->PrimitiveCount );
   }
}
```

The above code essentially describes the visibility determination process. For each leaf we loop through the number of render data structures stored there. We fetch each one and retrieve a pointer to the leaf bin that this structure is defined for. We then pass in the index count and primitive count members of the render data item into the leaf bin so that it knows this block of triangles needs to be rendered.

Of course, it is the leaf bin's AddVisibleData method which is responsible for either creating a new render batch or adding the passed triangle run to a render batch that has been previously created. The latter can only be done if the value of IndexCount passed into the function carries on consecutively from the indices of the last visible leaf that was visited and added to that render batch. The following code snippet shows what the basic responsibility of this function will be.

In this code, certain member variables of CLeafBin are assumed to exist. There is assumed to be an un-initialized array of RenderBatch structures large enough to store the maximum number of render batches that could be created during a visibility pass for a given subset. This allows us to avoid the need to resize the RenderBatch array of the leaf bin every time it needs to have more elements added or emptied at the start of a new visibility pass. We simply reset the leaf bin's m_nBatchCount member to zero at the

beginning of each visibility pass and increment it each time a new render batch needs to be created. A new render batch only needs to be created if the block of triangles described by the input parameters to the function does not follow on from those stored in the current render batch being compiled. Look at the code first and then we will discuss its basic operation. Understanding this logic is vitally important as it is the key collection process in our rendering system. It is our means of rendering as many consecutively arranged triangles in a single draw call as possible.

```
void CLeafBin::AddVisibleData( unsigned long IndexStart,
                               unsigned long PrimitiveCount )
{

    ULONG LastIndexStart      = m_nLastIndexStart,
          LastPrimitiveCount  = m_nLastPrimitiveCount;

    // Build up batch lists
    if ( LastPrimitiveCount == 0 )
    {
        // We don't have any data yet so just store initial values
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;

    } // End if no data
    else
    if ( IndexStart == (LastIndexStart + (LastPrimitiveCount * 3)) )
    {
        // The specified primitives were in consecutive order,
        // so grow the primitive count of the current render batch
        LastPrimitiveCount += PrimitiveCount;

    } // End if consecutive primitives
    else
    {
        // Store any previous data for rendering
        m_pRenderBatches[pData->m_nBatchCount].IndexStart     = LastIndexStart;
        m_pRenderBatches[pData->m_nBatchCount].PrimitiveCount = LastPrimitiveCount;
        m_nBatchCount++;

        // Start the new list
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;
    } // End if new batch

    // Store the updated values
    m_nLastIndexStart     = LastIndexStart;
    m_nLastPrimitiveCount = LastPrimitiveCount;
}
```

The leaf bin object is assumed to have member variables that are used temporarily by the visibility pass to track the last index start and primitive count passed into the function. These will be set to zero at the start of the visibility determination process for a given frame. At the end of the function the parameters passed in are stored in these member variables so that they are accessible to us when the next visible leaf is encountered and this method is called again. These variables represent our means of determining whether the index start parameter passed in describes the start of a run of triangles that follows on

consecutively from the data that was added to the render batch in the previous call. If so, these triangles can be added to the current render batch simply by incrementing the primitive count of the render batch currently being compiled. Otherwise, the new triangles we wish to add do not follow on consecutively in the leaf bin's index buffer, so we will need to end our current batch and start compiling a new render batch. Let us analyze the above function a few lines at a time so that we are sure we really understand how it works.

This function is called from a visible leaf when processing its render data structures. For example, if the RenderData structure for a visible leaf is linked to leaf bin 1, then the index count and primitive count stored in that structure are passed into the AddVisibleData method of the relevant leaf bin during the visibility pass for the tree. We are essentially informing the leaf bin that the current leaf has triangle data which is relevant to the leaf bin's attribute ID and should therefore be rendered.

```
void CLeafBin::AddVisibleData( unsigned long IndexStart,
                               unsigned long PrimitiveCount )
{

    ULONG LastIndexStart      = m_nLastIndexStart,
          LastPrimitiveCount  = m_nLastPrimitiveCount;
```

The IndexStart parameter will contain the position in the leaf bin's index buffer where this leaf's triangles begin and the primitive count will describe the number of triangles (starting from the IndexStart parameter) in the leaf bin's index buffer that belong to the calling leaf.

The first thing we do in the above code is fetch the LastIndexStart and LastPrimitiveCount members into local variables. These will both be set to zero at the start of the visibility pass and as such, will be zero when this function is first called when the first visible leaf is encountered for a given leaf bin. If these are zero, then it means we have not yet created any render batch structures in this visibility pass and this is the first time the function has been called. If this is the case, we simply store the passed index start and primitive count in the LastIndexStart and LastPrimitiveCount members. We are essentially starting a new batch collection process. At the bottom of this function we will then store these values in the m_nLastIndexCount and the m_nLastPrimitiveCount members so that when the next leaf which is found to be visible sends its data into this function, we have access to the information about the batch we are currently compiling. No other action is taken if this is the first time this function has been called for a given visibility pass, as shown in the following conditional code block.

```
    // Build up batch lists
    if ( LastPrimitiveCount == 0 )
    {
        // We don't have any data yet so just store initial values
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;
    } // End if no data
```

However, if LastPrimitiveCount does not equal zero then it means this is not the first time this method has been called for this leaf bin during the current visibility pass. In other words, this is not the first visible leaf that has been encountered that contains triangle data for this subset. If this is the case, then LastIndexStart will contain the location of the index that starts a consecutive run of triangles we are

currently collecting and LastPrimitiveCount will contain the number of triangles we have collected in the adjacent run so far. If the index start that has been passed into the function is equal to that of the start of the current batch we are building (LastIndexStart) plus the number of adjacent indices we have collected for this run so far (LastPrimitiveCount), it means the triangles of the leaf that called this method are arranged in the leaf bin's index buffer such that they continue the adjacent run of triangles we are currently trying to collect (i.e., the current batch we are compiling). When this is the case, we can simply increase the current primitive count for the current batch.

```
    else
    if ( IndexStart == (LastIndexStart + (LastPrimitiveCount * 3)) )
    {
        // The specified primitives were in consecutive order,
        // so grow the primitive count of the current render batch
        LastPrimitiveCount += PrimitiveCount;

    } // End if consecutive primitives
```

Finally, if none of the above cases are true it means that the index start passed in describes the location of a run of triangles for the leaf that does not exist immediately after the run of triangles we have compiled for the current batch. Thus, we have reached the end of a run of triangles that can be rendered in one batch. At this point, LastIndexStart and LastPrimitiveCount (with their values set in the previous call) will contain the start index and primitive count of a block of adjacent triangles that were being collected up until this call and therefore describe a block of triangles that can be rendered. We can add no more triangles to this batch. The current leaf data that was passed into this function cannot be added to this batch, so the previous index start and the primitive count collected so far are stored in a new RenderBatch structure, added to the leaf bin's RenderBatch array, and the number of render batches is increased. The LastIndexStart and LastPrimitiveCount members are then set to those values that were passed into the function, thus beginning a new cycle of trying to collect adjacent runs for a new batch. The remainder of the logic is shown below.

```
    else
    {
        // Store any previous data for rendering
        m_pRenderBatches[pData->m_nBatchCount].IndexStart     = LastIndexStart;
        m_pRenderBatches[pData->m_nBatchCount].PrimitiveCount = LastPrimitiveCount;
        m_nBatchCount++;

        // Start the new list
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;
    } // End if new batch

    // Store the updated values
    m_nLastIndexStart     = LastIndexStart;
    m_nLastPrimitiveCount = LastPrimitiveCount;
}
```

Understanding this process of collecting render batches is easier if we use some examples. Figure 15.2 shows the example arrangement we discussed earlier. We will now discuss the visibility processing step

performed when ProcessVisibility is called. In this example we will assume that all leaves except leaf 3 are visible. Just remember that no render batches exist in any leaf bins at the start of a visibility pass.

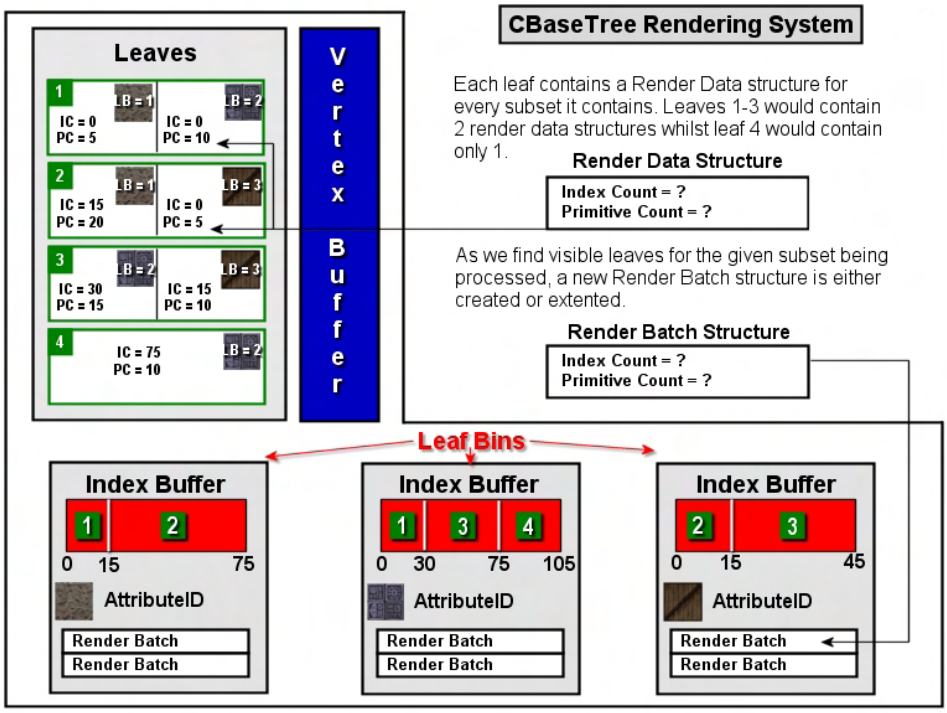


Figure 15.2

Leaf 1 is visited first and its first render data structure is processed. As this describes triangles that are contained in leaf bin 1, this leaf bin's AddVisibleData method is called and passed an index start of 0 and a primitive count of 5. As leaf bin 1 has not yet had this method called for any other leaf during the current visibility pass, we simply store 0 and 5 in its LastIndexStart and LastPrimitiveCount members. We then process the second render data structure in this leaf for leaf bin 2 and call its AddVisibleData method as well. An index start value of 0 is passed and a primitive count of 10 is passed. Since this is the first time leaf bin 2's AddVisibleData method has been called, the same thing happens. We simply store the passed index start and primitive count values in the LastIndexStart and LastPrimitiveCount members. At this point, no render batches have been created in any leaf bin, but in leaf bins 1 and 2, we have started compiling a batch.

Next we find that leaf 2 is visible, so we process its two render data structures. The first one is for leaf bin 1 again, so its AddVisibleData method is called and passed an index start of 15 and a primitive count of 20. However, this time when the method is called we detect that the index start passed in 15, is equal to the LastIndexStart value recorded (0) plus the last primitive count recorded (5) multiplied by 3. Therefore, we know that the triangles of leaf 2 stored in this leaf bin's index buffer follow on exactly from the triangles of leaf 1 that were passed in the previous call to this function. In this case, we can simply leave the LastIndexStart value of the leaf bin at its previous value (0) (which describes the starting location for the batch we are compiling) and increase the LastPrimitiveCount by the primitive count passed in. The value of LastPrimitiveCount is therefore increased from 5 (leaf 1's triangles) to 25 (adding leaf 2's 20 triangles). The current batch we are compiling now describes 25 triangles that can be rendered with a single draw call.

At this point in the discussion we will skip ahead to the end of the process as we can see that no further visible leaves exist that contain triangle data for this leaf bin. Therefore, at the end of the process we will simply create a single render batch for leaf bin 1 that has an index start of 0 and a primitive count of 25.

This render batch can be rendered with a single DrawIndexedPrimitive call and we can see that we have been able to merge the triangles from leaf 1 and leaf 2 that use this subset into a single render batch. Later, when the application calls ISpatialTree::DrawSubset and passes in the subset ID assigned to the first leaf bin, we can simply extract the index start and primitive count values from the leaf bin's only render batch and pass them as parameters to the DrawIndexedPrimitive call. This system of collecting adjacent runs into render batches allows us to lower the number of draw calls that will have to be made to render the triangles of a single leaf bin by consolidating adjacent runs into render batch instructions. Let us now continue with our example so that we can see where and why multiple render batches will sometimes need to be created for a given leaf bin.

When we left off in our example, we had just processed the first render data structure in leaf 2. When we process the second render data structure in this leaf (for leaf bin 3) we call leaf bin 3's AddVisibleData method and pass in an index start of 0 and a primitive count of 5. As this is the first leaf we have encountered that has data for leaf bin 3, this method simply copies the values of the past index start and primitive count parameters into leaf bin 3's LastIndexStart and LastPrimitiveCount members, thus starting the compilation of a new render batch..

Now we get to leaf 3 and determine that it is outside the frustum and therefore not visible. So we can skip it and its triangle data is never added to any leaf bins. We can once again skip ahead to the end of the process and see that in this instance, leaf bin 3 will have a single render batch generated for it with an index start of 0 and a primitive count of 5.

Finally, we get to the next visible leaf (leaf 4). This leaf contains 10 triangles that are stored in leaf bin 2 starting at location 75 in the index buffer. Therefore, we pass the values of 75 and 10 into leaf bin 4's AddVisibleData function. Now for the important part!

Leaf bin 2 has already been called during this visibility pass when leaf 1 was processed. Leaf 1 passed in a primitive count of 10 and a starting index 0. Therefore, the LastIndexStart member in this leaf bin will currently hold a value of 0 and the LastPrimitiveCount member will currently hold 10. We can see just by looking at the index buffer for leaf bin 2 in Figure 15.2 that leaf 4's triangles do not follow on from leaf 1's triangles. Leaf 3's triangles are inserted between them, but these triangles are not to be rendered because leaf 3 is not visible. In the code we just showed, we tested if the passed index start value described a consecutive run by testing to see if multiplying LastPrimitiveCount by 3 and adding it to LastIndexStart was equal to the index start value passed in. If they are the same, then the triangles we pass in continue on from those last added to the current batch we are compiling. Let us just perform a dry run of the function code shown above to make sure that this is definitely not considered an adjacent run:

```
if ( IndexStart == LastIndexStart + (LastPrimitiveCount * 3) ) )
{
  Increase LastPrimitiveCount
}
```

When this function is called by leaf 4, a value of 75 will be passed as the IndexStart parameter since this is where this leaf's triangles start in the index buffer for leaf bin 2. Furthermore, LastIndexStart will currently be set to 0 and LastPrimitiveCount will currently be set to 10 (leaf 1's primitive count) so this equates to the following test:

```
if ( 75 == 0 + (5 * 3)) )
{
  Increase LastPrimitiveCount
}
```

This equates to:-

```
if ( 75 == 15 )
{
  Increase LastPrimitiveCount
}
```

Clearly we can see that the above condition is not true and the triangles in leaf 4 cannot be rendered in the same draw call as those already collected from leaf 1. Therefore, the current value of LastIndexStart and LastPrimitiveCount will be stored as a new render batch in leaf bin 2 and LastIndexStart and LastPrimitiveCount will be set to the values stored in leaf 4's render data structure. When the process is over, these will also be added as a new render batch so that when visibility processing is complete, leaf bin 1 will have one render batch which renders the triangles from leaves 1 and 2 in a single draw call. Leaf bin 2 will have two render batches. One will render the triangles from leaf 1 and the other will render the triangles from leaf 4. This will have to be done with two separate draw calls. Leaf bin 3 will also contain a single render batch describing the triangles from leaf 2 that are visible and can be rendered with a single draw call.

Although this is a simple example, it is easy to imagine that during the tree traversal we will find many leaves that are visible and that have their triangle data arranged consecutively in the index buffers of the leaf bins to which they pertain. When such cases occur, we can render the triangles from multiple leaves with a single draw call instead of having to issue a DrawIndexedPrimitive call for each visible leaf that contains renderable data.

One thing that is immediately obvious about this system is that its success is dependant on the order in which we add the indices to the index buffer of each leaf bin during the BuildRenderData method. We can see in Figure 15.2 for example that because we process each leaf in the tree and add their indices to each leaf bin in the same order that we traversed those leaves during the visibility pass, we maximize the chance of collecting adjacent runs of triangles in multiple leaves. If we added leaf 2 to the index buffer first, followed by leaves 4, 3 and 1 (in that order) but still traversed the tree during the visibility pass in the 1,2,3,4 leaf order, we would get no adjacent runs and would end up having to perform a draw call for each leaf. That is, the number of render batches stored in each leaf bin would be equal to the number of leaves that contain polygons that use that leaf bin's attribute. This could be a very costly number of draw calls if every leaf was visible.

Luckily, we know that if every leaf is visible then our traversal methods will traverse that tree in the same order every time. Therefore, in the BuildRenderData method, one of the first things we will do is feed the entire scene's bounding box into the CollectLeavesAABB method. As we have passed the scene's bounding box to this function which encompasses all child leaves, the entire tree will be traversed and a list of all the leaves returned. However, the CollectLeavesAABB method would have added the leaves to the passed leaf list in the order in which the tree is traversed. That is, if the entire scene is visible during the ProcessVisibility call, this is the exact order in which the leaves will be

visited. Therefore, we will populate the leaf bins in that order by simply iterating through the leaf list returned from CollectLeavesAABB so that the indices of a leaf in the index buffer follow the indices of the previous leaf that will be visited during the traversal. In the case where the entire scene is visible, we will be adding visible data to each leaf bin's render batch system in the exact order that they are stored in the leaf bin's index buffer during the build phase. In this scenario, each leaf bin would create only one render batch that describes all the triangles stored in its index buffer. This means in the case where everything is visible, the number of draw calls will be reduced to the number of subsets/leaf bins in use by the tree.

Of course, when only some of the tree is visible there will be entire branches of the tree that are rejected from being further traversed and therefore, this will essentially end the render batch currently being compiled for each leaf. This is because the branch of the tree we have just rejected will describe a large section of the index buffer that we do not want to render and therefore ends any block of adjacent triangles we are trying to find. However, we at least know that if an entire branch of the tree is visible, because the visibility traversal order is the same order in which we added to the leaves to the index buffer of each leaf bin, all the leaves down that visible branch will be contained in the leaf bins as a continuous block of indices that can be represented as a single render batch and rendered with a single draw call.

15.2.2 Rendering the Tree

After the CBaseTree::ProcessVisibility call has been made by the application for a given frame update, the render batches for each leaf bin will have been constructed and will contain the triangles that need to be rendered for each leaf bin/subset. The application renders a given subset of the tree by calling the ISpatialTree::DrawSubset method. The code to such a function is shown below. Basically, it passes the drawing request on to the leaf bin's Render function. This is not the actual code from our lab project and all error checking has been removed. We only introduce it at this time to provide some insight into how the code will work. In the following function, the GetLeafBin method is exposed by CBaseTree and fetches the pointer to the leaf bin for the passed subset ID.

```
void CBaseTree::DrawSubset( unsigned long nAttribID )
{
    // Retrieve the applicable leaf bin for this attribute
    CLeafBin * pLeafBin = GetLeafBin( nAttribID );

    // Render the leaf bin
    pLeafBin->Render( m_pD3DDevice );
}
```

As the tree stores a leaf bin for each subset, this function simply fetches the leaf bin from its leaf bin array for the passed subset ID (attribute ID) and then issues a call to its Render function. Since the leaf bin has already compiled the render batches during the ProcessVisibility call, all it has to do is loop through each currently stored render batch and call the DrawIndexedPrimitive function to render the adjacent triangles described by each batch.

The code to such a function is shown below. Once again, this is not the actual code we will be using and it has had some error checking removed, but it does provide insight into the simple task assigned to the CLeafBin::Render method.

```
void CLeafBin::Render( LPDIRECT3DDEVICE9 pDevice )
{
    ULONG j;

    // Bin leaf bins index buffer to the device
    pDevice->SetIndices( pData->m_pIndexBuffer );

    // Render the leaves
    for ( j = 0; j < m_nBatchCount; ++j )
    {
          RenderBatch &Batch = m_pRenderBatches[j];

          // Render any data
          pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                         0,
                                         0,
                                         m_nVertexCount,
                                         Batch.IndexStart,
                                         Batch.PrimitiveCount );
    } // Next Batch
}
```

As you can see, this function is delightfully short and easy to follow. When the application requests that the tree render one of its subsets, we can see that it is really asking one of the leaf bins to render all of its visible triangles. The function first binds the leaf bin's index buffer to the device and then loops through each RenderBatch structure compiled during the ProcessVisibility pass for that leaf bin. For each render batch, it calls the DrawIndexedPrimitive method of the device passing in the start index and primitive count stored in the current render batch being processed. This describes the block of adjacent triangles in the index buffer represented by that render batch. In this example we will assume that m_nVertexCount is a member variable of CLeafBin and that it contains the number of vertices in the tree vertex buffer (i.e., the vertex buffer indexed into by all leaf bins).

Hopefully, you now see that our basic system is fairly logical and easy to understand. It is also fast because it allows us to work with static vertex and index buffers. Of course, this comes at the expense of potentially many more draw calls having to be issued than in the dynamic index buffer method, but it eliminates the abundance of memory copies that had to be performed to fill the dynamic index buffers during the visibility traversal. This proved to be more than a worthwhile tradeoff as our render batch system far outperformed the dynamic buffer approach in every test we ran.

15.2.3 Multiple Vertex Buffers

Thus far, we have been able to keep our system very simple with only a few structures needed to accomplish our goal. Essentially, we have leaves that contain information which describe their run of adjacent triangles in each leaf bin's index buffer, we have a leaf bin for each subset in use by the tree,

and we have RenderBatch structures that are built and collected in each leaf bin during a visibility pass. Finally, rendering a subset of the tree simply means asking the leaf bin associated with that subset to execute all its render batch instructions that were generated during the last visibility pass. Up until now we have assumed that all the vertices of all the polygons compiled into the tree can be stored in a single vertex buffer which is owned by the tree and indexed into by the single index buffer stored in each leaf bin. Unfortunately, vertex buffers cannot be of infinite size and furthermore, the maximum size of a vertex buffer which we can index into depends on the capabilities of the 3D device on which the application is running. One graphics card might be perfectly happy with using indices that reference vertices in a single buffer up to and beyond 16,000,000 (for example), while another card may insist on using vertex buffers containing less that 125,000. At the end of the day, we will need to make sure that the number of vertices we store in any vertex buffer is not greater than the maximum vertex index supported by the card. Therefore, if we find for example that we have 128,000 vertices comprising the static geometry of our spatial tree, but the system on which the application is currently running supports a maximum vertex buffer size of 64,000, we will have to store our 128,000 vertices in two vertex buffers (each holding 64,000). Each smaller vertex buffer is within the scope of the device's capabilities and as such, when we render the tree, we will have to render each vertex buffer (and their associated index buffers) separately.

This obviously has some ramifications for our system and forces us to add an additional layer of structures that we would rather not have to add. However, as we have discussed the system in the first part of this lesson without the multiple vertex buffer complication, adding it to our discussion now and discussing the changes that will have to be made will be much simpler to follow.

Because our tree may be using multiple vertex buffers to store its static geometry, we will have to make some fairly serious modifications to our system. First, during the BuildRenderData phase we will have to detect when the vertex buffer we are currently adding vertices to is full (i.e., it exceeds the maximum vertex indexing capabilities of the device) and create a new vertex buffer. To keep things simple, we will also create new index buffers for each leaf bin when this happens so that in each bin, we have separate index buffers for every vertex buffer being used by the tree. That is, if the tree is using five vertex buffers to store its geometry and a particular leaf bin contains triangles from each of these five vertex buffers, it will have five index buffers generated for it. So we will keep a logical pairing between vertex buffers and index buffers. We know when rendering the leaf bin that if it has five index buffers, we will have to loop through each, set the corresponding vertex and index buffers, and then render the triangles in that index buffer.

We will also need to know which vertex buffer a given leaf's RenderData structure is relative to. For example, we know that the RenderData structure in a leaf describes the index start and primitive count of one of its subsets. If its AttributeID is 2, this means it describes triangles in leaf bin 2. The problem is, leaf bin 2 may now have multiple index buffers, so we will need to store which index buffer / vertex buffer combination should be bound to the device in order to render the batch of triangles stored at that leaf. For example, we might imagine a case where leaf 1 has a RenderData structure that describes triangles in vertex buffer 1 and index buffer 1, but leaf 100's RenderData structure could contain exactly the same index start and primitive count values for the exact same subset but describe a run of adjacent triangles in the leaf bin's second vertex / index buffer pair. So we can see that each leaf's RenderData structure will have to store the numerical index of the vertex/index buffer pair that contains the vertices

of the triangles it describes. We might imagine at this stage then that the RenderData structure would look something like this:

```
struct RenderData
{
   unsigned long IndexCount;
   unsigned long PrimitiveCount
   unsigned long VBIndex;
   unsigned long AttributeID;
   CLeafBin     * pLeafBin;
}
```

Looking at this structure we can see that it contains not only the start index of where its indices start in the associated leaf bin's index buffer and the number of triangles stored there, but also contains the numerical index of the vertex buffer/index buffer combination that store the triangle data. We can see in this example how it might also contain the attribute ID which describes the subset/leaf bin this RenderData structure is associated with and a pointer to the leaf bin that stores triangles for this subset.

Storing the leaf bin pointer in the RenderData structure is very useful for speeding up the visibility process. When processing the RenderData structure for a given visible leaf, we have to pass its index count and primitive count into the leaf bin's AddVisibleData method so that it can be added to an existing render batch or used to start a new batch. By having this pointer stored in the data structure, we negate the need to search for the leaf bin with a matching attribute ID. We really need to make the tree traversal and visible polygon collection process as quick as possible, so we do not want to waste time looping through the leaf bin array finding the leaf bin the triangles in this RenderData structure should be added to. Therefore, we will find this during the building process and store the leaf bin's pointer in the RenderData structure for immediate access.

We can now see that the RenderData structure not only describes the number of triangles in a leaf bin and their location within the leaf bin's index buffer, it also describes to the leaf bin which vertex buffer should be bound to the device and which of the leaf bin's index buffers the triangle run is contained in. For example, if VBIndex were set to 3, this would inform the leaf bin that the vertices of these triangles are stored in the third vertex buffer in the tree's vertex buffer array and that its indices are stored in the leaf bin's third index buffer.

Unfortunately the provisions made here for the multiple vertex buffer case do not cover all scenarios we may encounter. For example, imagine the following situation during the BuildRenderData method:

We are currently visiting leaf 10 and we are processing the polygons in that leaf that use subset 6. Assume that there are 20 polygons in this leaf that use subset 6 and that up until this point we have been adding the vertex data to a single vertex buffer. Now imagine that as we add the first 10 triangles of this leaf's subset 6 polygons to the vertex buffer (and its indices to leaf bin 6's index buffer) we discover that we have filled up the vertex buffer. We have encountered a situation where we have to switch to a new vertex buffer (and create a new index buffer to go with it in leaf bin 6) halfway though processing the polygons for a single subset. Although this might happen only rarely, it is certainly possible that triangles from the same leaf that share the same subset are split over multiple buffers. In such a case, the RenderData structure should describe not only the vertex and index buffer to use for a given subset in

that leaf, but should describe all the vertex buffers/index buffers that contain subset 6 in that leaf. If we take this into account, we end up with a RenderData structure stored at each leaf that looks like so:

```
struct RenderElement
{
  unsigned long IndexStart;
  unsigned long PrimitiveCount
  unsigned long VBIndex;
};


struct RenderData
{
   RenderElement *pRenderElements;
   unsigned long ElementCount;
   unsigned long AttributeID;
   CLeafBin     * pLeafBin;
}
```

Let us stop and think about what information we are actually storing here. The RenderElement structure now stores the index count and primitive count for a single vertex/index buffer combination for a given subset. In the above example where the 20 polygons that used subset 6 in a leaf were spread over two vertex/index buffers, the RenderData structure in that leaf for subset 6 would contain two render elements in its array and would describe two runs of triangles in two of the leaf bin's index buffers. The VBIndex of the first would be 0 describing the first vertex buffer and the VBIndex of the second would be 1. When we reach a visible leaf during the visibility process, the leaf bin would now need to be passed the VBIndex of the render element alongside the index start index and primitive count of each element stored there. This is because, now that the leaf bin maintains an array of index buffers (one for each primary vertex buffer being used by the system), an array of RenderBatch lists will now need to be stored also. That is, the leaf bin will need to compile a render batch list for each vertex/index buffer pair in use by the leaf bin.

The following code shows what happens when a visible leaf is encountered during the ProcessVisibility call and the leaf's SetVisible function is called.

```
void CBaseLeaf::SetVisible( bool bVisible )
{
    ULONG                 i, j;
    RenderElement       * pElement;
    CLeafBin            * pLeafBin;
    RenderData          * pData;

    // Flag this as visible
    m_bVisible = bVisible;

    // If we're being marked as visible, inform the renderer
    if ( m_bVisible && m_nRenderDataCount > 0 )
    {
        // Loop through each renderable set in this leaf.
        for ( i = 0; i < m_nRenderDataCount; ++i )
        {
```

```
            pData    = &m_pRenderData[i];
            pLeafBin = pData->pLeafBin;

            // Loop through each element to render
            for ( j = 0; j < pData->ElementCount; ++j )
            {
                pElement = &pData->pElements[j];
                if ( pElement->PrimitiveCount == 0 ) continue;

                // Add this to the leaf bin
                pLeafBin->AddVisibleData( pElement->VBIndex,
                                          pElement->IndexStart,
                                          pElement->PrimitiveCount );
            } // Next Element

        } // Next RenderData Item

    } //  End if visible
}
```

The function first stores the passed visibility Boolean in the leaf's member variable which describes the leaf as either being visible or non visible. If the leaf has been flagged as visible then we need to add the triangles of every RenderData item stored there (one for each subset contained in the leaf) to their associated leaf bins. Notice how we loop through each RenderData structure stored in the leaf's array to add its data to its attached leaf bin. However, the triangles described by a single RenderData structure may be split over multiple vertex buffers (and multiple index buffers in the leaf bin). Therefore, for each RenderData structure we then loop through each RenderElement stored there. There will usually only be a single RenderElement structure stored in a RenderData structure. The exception to the rule occurs when, during the adding of this subset's polygon data to the vertex buffer, a new buffer had to be created such that the RenderData of a leaf for a given subset spanned vertex buffer boundaries. For each render element, we then call the leaf bin's AddVisibleData member. We looked at a single vertex buffer version of this function earlier, but let us now see how that function might look now that multiple lists of render batches will need to be maintained in the leaf bin for each vertex/index buffer pair.

Because the leaf bin no longer contains only a single index buffer and a single render batch list, we will need to introduce an additional structure in the leaf bin, which in this code is called CLeafBinData. This is a simple structure that pairs an index buffer with one of the tree's vertex buffers and stores its associated RenderBatch list during the visibility traversal. The CLeafBin class would now have just the following members (methods not shown):

```
class CLeafBin
{
    unsigned long  m_nAttribID;
    CLeafBinData **m_ppBinData;
    unsigned char  m_nVBCount;
};
```

The first member would describe the subset ID for which this leaf bin houses indices and compiles render batches for. The second is a pointer to an array of CLeafBinData objects. It is a CLeafBinData object that contains the pointers to an index buffer and vertex buffer pair and contains the RenderBatch

list for that buffer pair. As an example, if this leaf bin stored triangles for subset 6, and the vertices of polygons in the tree that use subset 6 are distributed across five vertex buffers, this leaf bin's m_nVBCount member would be set to 5 and would describe the number of elements in the CLeafBinData array. Each CLeafBinData structure then describes the index buffer and render batches used to render the triangles from those buffers.

The members of the CLeafBinData object would be as follows:

```
class CLeafBinData
{
   LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer;
   LPDIRECT3DINDEXBUFFER9  m_pIndexBuffer;
   unsigned long           m_nLastIndexStart;
   unsigned long           m_nLastPrimitiveCount;
   RenderBatch            *m_pRenderBatches;
   unsigned long           m_nBatchCount;
};
```

There is an important distinction to make between the vertex buffer pointer and the index buffer pointer in this structure. The vertex buffers are created and managed by the tree, not the leaf bins, so this is a pointer to a tree owned vertex buffer. There may be CLeafBinData items in multiple leaf bins that all point to this same vertex buffer as they all have triangles stored within it. The index buffers however are created and managed by the leaf bin and will not be shared by any other bins. That is, an index buffer contains the triangles for a specific leaf bin that are contained in a specific vertex buffer. Notice also that in our original discussion it was the leaf bin that managed the list of RenderBatch structures during the visibility pass. This is now stored in the CLeafBinData structure as we will need to compile render batch lists for each index/vertex buffer used by the bin. This is because we will have to execute the render batches for each index buffer only after we have bound it to the device along with its associated vertex buffer. Finally, the m_nLastPrimitiveCount and m_nLastIndexStart members have also been moved from the leaf bin into the CLeafBinData structure. These members were used during render batch building to remember the settings for the current batch being compiled between function calls. As we now have multiple index buffers in a single leaf bin, we will have to maintain multiple render batch lists and therefore, will need to store these 'm_nLast…' members on a per batch list basis.

We saw just a moment ago how the CBaseLeaf::SetVisible method calls the CLeafBin::AddVisibleData method to add the triangles of each RenderElement in each RenderData structure stored in the leaf. We had a look at a single vertex buffer version of this function earlier. Let us now see what it might look like with multiple vertex buffer support in the leaf bins.

```
void CLeafBin::AddVisibleData( unsigned char VBIndex,
                               unsigned long IndexStart,
                               unsigned long PrimitiveCount )
{
   CLeafBinData * pData = m_ppBinData[ VBIndex ];

   ULONG LastIndexStart       = pData->m_nLastIndexStart,
         LastPrimitiveCount   = pData->m_nLastPrimitiveCount;

   // Build up batch lists
```

```
   if ( LastPrimitiveCount == 0 )
   {
        // We don't have any data yet so just store initial values
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;

   } // End if no data
   else if ( IndexStart == (LastIndexStart + (LastPrimitiveCount * 3)) )
   {
        // The specified primitives were in consecutive order
        LastPrimitiveCount += PrimitiveCount;

   } // End if consecutive primitives
   else
   {
   // Store any previous data for rendering
   pData->m_pRenderBatches[pData->m_nBatchCount].IndexStart= LastIndexStart;
   pData->m_pRenderBatches[pData->m_nBatchCount].PrimitiveCount=LastPrimitiveCount;
   pData->m_nBatchCount++;

        // Start the new list
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;

   } // End if new batch

   // Store the updated values
   pData->m_nLastIndexStart     = LastIndexStart;
   pData->m_nLastPrimitiveCount = LastPrimitiveCount;
}
```

As you can see, the changes are subtle as the render batches are now compiled in the CLeafBinData structures instead of the CLeafBin object's themselves. The function's first parameter is the index of the tree's vertex buffer that the passed index start and primitive count value pertain to. We use the vertex buffer index to fetch the correct CLeafBinData structure from the leaf bin's array (there will be one for each vertex buffer being used by the tree) and from that point on, we are simply growing or starting new render batches in the CLeafBinData structures instead of in the leaf bin itself.

After the visibility process has been performed on the tree, each leaf bin may contain multiple render batches in each CLeafBinData structure. Each structure in this array describes the batches of the visible triangles that need to be rendered for a specific vertex/index buffer pair.

At this point, the application will choose to render the subsets of the tree. We saw earlier that the CBaseTree::DrawSubset call really just forwards the render request on the leaf bin that matched the passed subset ID. We did examine a version of a CLeafBin->Render function, but it lacked multiple vertex buffer support. Here is the new version of the function that instructs the leaf bin to render all render batches associated with all vertex/index buffers it contains.

```
void CLeafBin::Render( LPDIRECT3DDEVICE9 pDevice )
{
    ULONG i, j;
```

```
    // Loop through each vertex buffer
    for ( i = 0; i < m_nVBCount; ++i )
    {
        CLeafBinData * pData = m_ppBinData[ i ];
        if ( !pData ) continue;

        // Set the stream sources to the device
        pDevice->SetStreamSource( 0, pData->m_pVertexBuffer, 0, sizeof(CVertex) );
        pDevice->SetIndices( pData->m_pIndexBuffer );

        // Set the FVF
        pDevice->SetFVF( VERTEX_FVF );

        // Render the leaves
        for ( j = 0; j < pData->m_nBatchCount; ++j )
        {
            RenderBatch & Batch = pData->m_pRenderBatches[j];

            // Render any data
            pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                           0,
                                           0,
                                           pData->m_nVertexCount,
                                           Batch.IndexStart,
                                           Batch.PrimitiveCount );

        } // Next element

    } // Next Vertex Buffer
}
```

This function has the task of looping through each CLeafBinData structure in its array, binding its vertex and index buffers to the device and then looping through its render batches and rendering each one. Remember, the leaf bin's m_nVBCount member describes the number of vertex buffers in use by the tree and therefore describes the size of each leaf bin's CLeafBinData array. If the tree stored its geometry in five vertex buffers, but the leaf bin only contained triangles stored in the first vertex buffer, there would still be five CLeafBinData objects in its array, although the other four would have no render batches and no index buffers created or stored in them.

So, we loop through each CLeafBinData structure and for each one we process, we bind its vertex and index buffer pair to the device. We then loop through its render batches and call DrawIndexedPrimtive to render each batch. At the end of the outer loop we will have rendered all visible geometry in this leaf bin.

Although this sounds like a complicated system, it will become easier to grasp when we start to cover the actual code in the next section. Seeing how the leaf bins are constructed during the BuildRenderData call should clear up a lot of unanswered questions you may have at this point. The goal of this first section has been to provide you with some insight as to how the system will work and the various structures that will be used to implement it. This should make covering the rather tedious source code a lot easier.

It is now time to start discussing the real source code in the CBaseTree render system. As we have often done, we will cover the source code from a bottom up perspective. That is, we will look at the changes we will have to make at the CBaseLeaf level and filter those changes up until we finally cover the code changes to CBaseTree itself. To not do so would be pointless. For example, if we were to cover the CBaseTree::BuildRenderData method first, you will see that it spends most of its time calling member functions of CBaseLeaf and CLeafBin to accomplish its tasks. By looking at these support objects first, we can make sure that when we do finally get to examining the CBaseTree code, we understand all the function calls it is making and the support structures used.

15.3 CBaseLeaf - Adding Rendering Support

In the previous lesson we discussed much of the CBaseLeaf implementation and we listed and examined all code that related to building and querying the tree. In this section we will look at CBaseLeaf methods and member variables that were not discussed in the previous lesson as they pertained to the render system.

As the previous section indicated, part of the job of the CBaseTree::BuildRenderData method will be to store at each leaf a RenderData structure for each subset contained in that leaf. That is, if the polygon data in a given leaf uses three attributes (i.e., belongs to three different subsets), there will be an array of three RenderData items stored at that leaf.

We also saw that because we need to support multiple vertex buffers, each RenderData item cannot just store a single index start and primitive count since the associated leaf bin will also need to know in which of the tree's vertex buffers the triangles are contained. This also indicates which index buffer that the leaf will have stored these triangles in within the leaf bin. The RenderData structure represents a single subset within a leaf. It contains a pointer to the associated leaf bin that collects triangles for that subset and also contains an array of render elements (Element structures).

15.3.1 The RenderData Structure

The RenderData structure is shown below along with the Element structure which is contained within its namespace. Its definition is contained in CBaseTree.h. The RenderData structure is actually defined inside the CBaseLeaf namespace, but for the sake of clarity in this discussion, we show it outside that namespace.

```
struct RenderData
{
        struct Element
        {
            unsigned long   IndexStart;         // Index of this leaves first tri
            unsigned long   PrimitiveCount;     // Number of Tris
            unsigned char   VBIndex;            // vertex Buffer Index
        };
```

```
        unsigned long AttributeID;    // The attribute ID of this render data item
        CLeafBin    * pLeafBin;       // Pointer to the actual leaf bin
        Element     * pElements;      // The actual render data element array
        unsigned long ElementCount;   // The number of elements stored here.
};
```

Let us discuss the four members of this structure.

unsigned long AttributeID
This member stores the subset ID of the triangles in the leaf that this RenderData structure represents. This will match the attribute ID of the leaf bin pointed at by its pLeafBin member.

CLeafBin * pLeafBin
This member stores a pointer to the leaf that is to collect the triangles in this RenderData structure during a visibility pass if the leaf in which the RenderData structure is contained is deemed visible. We store the pointer to the leaf bin in the RenderData structure so that we can easily call its AddVisibleData method to send it the render elements of this RenderData structure.

Element * pElements
This is a pointer to an array of Element structures. Each element describes a run of triangles that needs to be added to the leaf bin during a visibility pass, along with a vertex buffer index. The vertex buffer index informs the leaf bin's AddVisibleData methods which render batch list the triangles in that element should be added to. Most of the time, a single RenderData structure will contain only one Element in this array which describes (to the associated leaf bin) which vertex buffer/index buffer pair these triangles are associated with and therefore, which render batch list the triangles should be added to in that leaf bin.

The only time when there will exist more than one Element in a leaf's RenderData structure will be if, during the BuildRenderData method, a vertex buffer switch had to be made partway through adding a leaf's subset to a leaf bin. For example, we know that if we have 50 polygons that use subset 5, during the build phase the vertices of these polygons will need to be added to the tree's vertex buffer and the indices of these 50 triangles should be added to the leaf bin's index buffer. Under normal circumstances were no complications to occur, we would then store (inside the RenderData structure for that leaf's subset) a primitive count of 50 and an index start describing where this leaf's run of triangles start in that leaf bin's index buffer. However, what if after adding only 25 triangles during the build process, the vertex buffer becomes full and the remaining 25 of this leaf's subset 5 polygons have to be added to a new vertex buffer? The leaf bin will be informed when the second 25 triangles are added that these exist in the second vertex buffer, not the first. The leaf bin would then create a second index buffer that will be used to index the triangles in the second vertex buffer. If we go back to the leaf and subset in question, we now have a RenderData structure for subset 5 in a leaf that is spread over two vertex and index buffers. Therefore, in such a situation two Element structures will be stored in its RenderData structure's pElements array. The first element will describe the array of subset 5 triangles in the first vertex/index buffer and the second element will describe the second run of subset 5 triangles in the second vertex/index buffer. As we discussed, during the visibility pass, each element will be passed to the leaf bin which will maintain render batch lists for each vertex/index buffer combination it uses.

unsigned long ElementCount
This member describes the number of elements stored in the previously described array. As discussed, this will usually be 1 unless the polygons in this leaf that belong to the subset represented by this RenderData structure span multiple vertex buffers.

15.3.2 CBaseLeaf – The Source Code

CBaseLeaf is no stranger to us as it was covered quite heavily in the previous chapter. In this chapter however we will cover its methods and members pertaining to the rendering system. Although we show the entire class below contained CBaseTree.h (minus dynamic object support which will be added later in the lesson), you will see that there are only a handful of new methods and members that we did not cover in the previous lesson. These are highlighted in bold.

```
class CBaseLeaf : public ILeaf
{
public:

    // Constructors & Destructors for This Class.
    virtual ~CBaseLeaf( );
             CBaseLeaf( CBaseTree * pTree );

    // Public Virtual Functions for This Class (from base).

    virtual bool            IsVisible           ( ) const;

    virtual unsigned long   GetPolygonCount     ( ) const;
    virtual CPolygon *      GetPolygon          ( unsigned long nIndex );
    virtual unsigned long   GetDetailAreaCount  ( ) const;
    virtual TreeDetailArea* GetDetailArea       ( unsigned long nIndex );
    virtual void            GetBoundingBox      ( D3DXVECTOR3 & Min,
                                                  D3DXVECTOR3 & Max ) const;

    // Public Functions for This Class.
    void                    SetVisible          ( bool bVisible );

    void                    SetBoundingBox      ( const D3DXVECTOR3 & Min,
                                                  const D3DXVECTOR3 & Max );
    bool                    AddPolygon          ( CPolygon * pPolygon );
    bool                    AddDetailArea       ( TreeDetailArea * pDetailArea );

    RenderData            * AddRenderData       ( unsigned long nAttribID );
    RenderData::Element   * AddRenderDataElement( unsigned long nAttribID );
    RenderData            * GetRenderData       ( unsigned long nAttribID );


protected:


    // Protected Structures, Enumerators and typedefs for This Class.
```

```
    typedef std::vector<CPolygon*>          PolygonVector;
    typedef std::vector<TreeDetailArea*>    DetailAreaVector;


    // Protected Variables for This Class

    PolygonVector       m_Polygons;         // Array of leaf's polygon pointers
    DetailAreaVector    m_DetailAreas;      // Array of detail area pointers.
    bool                m_bVisible;         // Is this leaf visible or not?
    D3DXVECTOR3         m_vecBoundsMin;     // Minimum bounding box extents
    D3DXVECTOR3         m_vecBoundsMax;     // Maximum bounding box extents.

    RenderData         *m_pRenderData;      // Renderable data information
    unsigned long       m_nRenderDataCount; // Number of render data items .

    CBaseTree          *m_pTree;            // The tree to which this leaf belongs.
};
```

There are only three new members that have been added to the leaf which are used to store the RenderData structures.

RenderData *m_pRenderData
This array will contain a RenderData structure for each subset of polygons that exist in the leaf. The RenderData structure was discussed above and contains the runs of triangles (called elements) stored in the leaf bin's index buffers. If a leaf contains polygons that belong to five different subsets, the leaf will store five RenderData structures in this array. We can think of each RenderData structure as being an input of visible triangles into a specific leaf bin when the leaf is found to be visible.

unsigned long m_nRenderDataCount
This member describes the number the RenderData structures in the above array. This implicitly tells us the number of subsets contained in this leaf and the number of leaf bins we will have to provide render elements for if this leaf is found to be visible.

bool m_bVisible
This Boolean is set to either true or false by the visibility determination process. If the ProcessVisibility method of the spatial tree determines that this leaf is inside the view frustum, it will be set to true; otherwise it will be set to false.

Notice that CBaseLeaf also has several new methods which relate to the visibility system. These methods will never be called by the application, as they are used by the tree when building and preparing the render data and processing tree visibility. We will discuss each of these methods next.

IsVisible – CBaseLeaf

This method is an exception to the rule with respect to the other leaf visibility methods in that it will often be used by the application to query the visibility status of a leaf. For example, the application may send an AABB down the tree to collect all leaves that fall within that bounding volume. The CollectLeavesAABB method would be used to fill a leaf list with these CBaseLeaf pointers. The application could then iterate through the returned leaf list and call the IsVisible method to determine if that leaf is currently visible and needs to be processed.

The method itself is a simple one line function that returns the value of its m_bVisible member variable. It will contain the visibility status of the leaf as calculated in the last visibility pass through the tree (i.e., the most recent call to CBaseTree::ProcessVisibility).

```
bool CBaseLeaf::IsVisible( ) const
{
    return m_bVisible;
}
```

AddRenderData – CBaseLeaf

The CBaseLeaf::AddRenderData method should never be called by the application. It is only called by the spatial tree during the BuildRenderData call when the data is being prepared in its renderable form. It is at this stage (just after the tree has been built) that each leaf is processed and the RenderData structures are generated and stored in each leaf (one for each subset contained in that leaf).

Since the leaf stores its RenderData structures in an array, this array will have to be grown every time we wish to add a new RenderData structure to it during the building of the render data. Remember, at the start of the BuildRenderData call, no leaves will contain any RenderData structures. These will need to be allocated and added to a leaf's RenderData array as and when we examine the polygons of that leaf and find a polygons in that leaf with an attribute ID which we have not yet generated a RenderData structure for. The CBaseTree::BuildRenderData method will call this method when it would like to make room in the leaf's RenderData array for a new structure. The function is passed the attribute ID of the subset that this RenderData structure represents in the current leaf. The function will resize the leaf's RenderData structure array and store the attribute ID in the new RenderData structure. A pointer to this new element in the array will then be returned to the function so that it can be used to populate the new RenderData structure with render elements.

This code is like most we have seen that resizes arrays. It first allocates a new empty array of RenderData structures that is large enough to store the number of elements currently in the RenderData array plus the one new element we wish to add. We then copy over all the data from the previous array into the new array before releasing the old array and assigning the leaf's pointer to point at this new array instead. We also store the passed attribute ID in the newly added RenderData structure so that we know which subset set this RenderData structure will store polygons for.

```
CBaseLeaf::RenderData * CBaseLeaf::AddRenderData( unsigned long nAttribID )
{
    RenderData * pBuffer, * pData;

    // First allocate space for all the element items
    pBuffer = new RenderData[ m_nRenderDataCount + 1 ];
    if ( !pBuffer ) return NULL;

    // Any old data?
    if ( m_nRenderDataCount > 0 )
    {
        // Copy over the old data, and release the old array
        memcpy( pBuffer, m_pRenderData, m_nRenderDataCount * sizeof(RenderData) );
        delete []m_pRenderData;

    } // End if any old data

    // Get the element we just created
    pData = &pBuffer[ m_nRenderDataCount ];

    // Clear the new entry
    ZeroMemory( pData, sizeof(RenderData) );

    // Store the attribute ID, it's used to look up the render data later
    pData->AttributeID = nAttribID;

    // Store the new buffer pointer
    m_pRenderData = pBuffer;
    m_nRenderDataCount++;

    // Return the item we created
    return pData;
}
```

Notice that when this new RenderData structure is added to the array it is initially empty. It contains no Elements in its pElements array, which means initially it will be linked to no leaf bin and will not contain any triangles. The calling function will use the returned pointer to set its leaf bin pointer and add Elements to it.

GetRenderData - CBaseLeaf

Although the application should never need to call a function such as this, it will be used by the BuildRenderData method when it wishes to retrieve a pointer to one of the leaf's RenderData structures. A classic example of this is when the BuildRenderData method is processing the polygons in a given leaf. If it finds a polygon that uses subset 5 for example, it will call the GetRenderData method passing in an attribute ID of 5. If a valid pointer is returned, then it means we have already processed triangles in this leaf for the same subset and as such, its RenderData structure has already been allocated. When this is the case, we can just increment the primitive count of the RenderData structure and continue on to the next polygon in the leaf. If a NULL pointer is returned, it tells the BuildRenderData method that we are currently processing a polygon in a leaf that belongs to a subset we have not yet found in that leaf. This

means that there will not currently be a RenderData structure allocated for it. When this happens, the BuildRenderData method can call the CBaseLeaf::AddRenderData method to add a new RenderData structure to the leaf's array. The current index start and primitive count of this polygon (and any future ones we find in this leaf which share the same subset as this polygon) can then be appropriately stored.

The key for the search is the subset ID (attribute ID) of the RenderData structure you are looking for within this leaf. The function loops through the leaf's array of RenderData structures searching for one with a matching attribute ID. If one is found, its pointer is returned. Otherwise, the loop plays out to its conclusion and we return NULL. This indicates that no RenderData item currently exists in the leaf which matches the passed attribute ID.

```
CBaseLeaf::RenderData * CBaseLeaf::GetRenderData( unsigned long nAttribID )
{
    ULONG i;

    // Search for the correct render data item
    for ( i = 0; i < m_nRenderDataCount; ++i )
    {
        if ( m_pRenderData[i].AttributeID == nAttribID ) return &m_pRenderData[i];
    } // Next RenderData Item

    // We didn't find anything
    return NULL;
}
```

You will see many of these methods being used when we cover the CBaseTree::BuildRenderData method later in the lesson.

AddRenderElement - CBaseLeaf

This method is another method that will never be called by the application but will be used by the CBaseTree::BuildRenderData method while constructing the RenderData structures at each leaf.

As discussed previously, each RenderData structure stored in a leaf will represent that leaf's polygon contribution for a single subset. Each RenderData structure is used to describe a block of triangles in the associated leaf bin's index buffer(s). Usually, each RenderData structure will store a single Element where each element represents the block of triangles in a given leaf bin's index buffer. However, if when adding the vertices of a given leaf's subset, the vertex buffer is found to be full, a new vertex buffer will be generated and that leaf's subset will be split over multiple vertex buffers. When this is the case, we must store multiple Elements in the RenderData structure so that we can inform the associated leaf bin during the visibility pass about which vertex/index buffers store all the triangles for this leaf. For example, if this leaf has polygons that belong to subset 10, but they are spread over three different vertex buffers, the leaf's RenderData structure for subset 10 will contain three Elements. Each Element describes the vertex buffer and index buffer this run of triangles is stored in and the location and number of indices in those buffers.

Because we never know before building how many Elements we will need to store in each leaf, we will need a way for the BuildRenderData method to add new Elements to a leaf's RenderData item structure as and when we encounter a situation where one has to be added. The AddRenderElement method simply resizes the Element array of the leaf's RenderData structure that is associated with the passed subset ID making room at the end for a new Element structure. A pointer to this new Element is returned to the calling function so that it can populate the new Element structure with index start and primitive count information for a given vertex buffer. We will see how this function is used later.

The method is passed a subset ID as its only parameter. The Leaf's GetRenderData method is then called to retrieve a pointer to the RenderData structure within the leaf that represents triangles assigned with the passed attribute.

```
CBaseLeaf::RenderData::Element * CBaseLeaf::AddRenderDataElement
                                                ( unsigned long nAttribID )
{
    RenderData::Element * pBuffer, * pElement;
    RenderData * pData = GetRenderData( nAttribID );
    if ( !pData ) return NULL;
```

If the GetRenderData method returned NULL, it means the caller is trying to add an Element to the array of a RenderData structure which has not yet been created. In this case, the function returns NULL. This informs the caller that there currently exists no RenderData structure in the leaf associated with the passed subset ID.

At this point we allocate a new Element array (notice it is within the RenderData namespace) large enough to store any Elements that may already be in the array plus the new one we wish to add. If there are any elements already in the previous array of the RenderData structure, we copy them over into the new array we have just allocated. We then delete the old array previously pointed at by the RenderData::pElements pointer.

```
    // First allocate space for all the element items
    pBuffer = new RenderData::Element[ pData->ElementCount + 1 ];
    if ( !pBuffer ) return NULL;

    // Any old data?
    if ( pData->ElementCount > 0 )
    {
        // Copy over the old data, and release the old array
        memcpy( pBuffer,
                pData->pElements,
                pData->ElementCount * sizeof(RenderData::Element) );

        delete []pData->pElements;

    } // End if any old data
```

At this point we store a pointer to the new Element structure we added at the end of the new array so we can return the structure to the caller.

```
    // Get a pointer to the new element
    pElement = &pBuffer[ pData->ElementCount ];
```

The new Element structure at the end of the array is un-initialized, so we clear its memory.

```
    // Clear the new entry
    ZeroMemory( pElement, sizeof(RenderData::Element) );
```

We then assign the RenderData structure's pElements pointer to point at our new array (as the old one has been deleted) and increase the RenderData::ElementCount member variable to correctly reflect the number of elements in the array. Finally, we return the pointer to the new element at the end of this array which the caller will then use to populate with meaningful data.

```
    // Store the new buffer pointer
    pData->pElements = pBuffer;
    pData->ElementCount++;

    // Return the element we created
    return pElement;
}
```

SetVisible - CBaseLeaf

We have seen a few simplified versions of this function during our initial discussions of the render system, but now we will see the real thing. Before we look at the code, you should be aware of a new member that has been added to CBaseTree.

We will see later that CBaseTree now has an additional LeafList member variable which is declared as shown below:

Excerpt from CBaseTree.h (CBaseTree class)

```
LeafList                    m_VisibleLeaves;
```

Remembering that LeafList is a type definition for an STL list that contains ILeaf pointers, the name of this member should imply what it is used to store.

When the visibility pass is performed on the tree, a traversal method in the derived class will be used to walk through the tree finding visible leaves. At the very start of this visibility traversal (which is performed with each frame update), the m_VisibleLeaves list will be emptied. Once a visible leaf has been found during traversal, the leaf's SetVisible method will be called. This function will set the leaf's visibility Boolean to true. It will also loop through every RenderData structure stored in the leaf and add each one's Elements to their associated leaf bins as we discussed earlier. All of this code has been covered earlier and is shown below.

```
void CBaseLeaf::SetVisible( bool bVisible )
{
```

```
    ULONG                  i, j;
    RenderData::Element * pElement;
    CLeafBin            * pLeafBin;
    RenderData          * pData;

    // Flag this as visible
    m_bVisible = bVisible;

    // If we're being marked as visible, inform the renderer
    if ( m_bVisible && m_nRenderDataCount > 0 )
    {
        // Loop through each renderable set in this leaf.
        for ( i = 0; i < m_nRenderDataCount; ++i )
        {
            pData    = &m_pRenderData[i];
            pLeafBin = pData->pLeafBin;

            // Loop through each element to render
            for ( j = 0; j < pData->ElementCount; ++j )
            {
                pElement = &pData->pElements[j];
                if ( pElement->PrimitiveCount == 0 ) continue;

                // Add this to the leaf bin
                pLeafBin->AddVisibleData( pElement->VBIndex,
                                          pElement->IndexStart,
                                          pElement->PrimitiveCount );
            } // Next Element

        } // Next RenderData Item

    } //  End if visible
```

However, the final part of this function is new. If the visibility status of the tree is set to true, the leaf adds its own pointer to the tree's m_VisibleLeaves list using a new method of CBaseTree called AddVisibleLeaf. This is a simple method which just wraps the adding of the passed leaf pointer to the tree's visible leaf list. The remainder of this function is shown below.

```
    // Update tree object's if we're visible
    if ( m_bVisible )
    {
        // // Add this leaf to the tree's visible leaf list
        m_pTree->AddVisibleLeaf( this );

    } // End if we are visible
}
```

After the visibility process has been performed for a given frame update, and the SetVisible method has been called for all visible leaves, the spatial tree's m_VisibleLeaves list will contain pointers for all leaves which are currently visible. The application can fetch this list by calling the spatial tree's GetVisibleLeaves method.

It is very useful for the application to be able to ask the tree for a list of currently visible leaves. For example, the application can loop through these leaves and only bother rendering the dynamic objects contained in those leaves. Any dynamic object which is stored in non-visible leaves will simply be ignored by the rendering loop. After we have discussed the rendering system for the static geometry of the spatial tree, we will discuss how the application can assign its own dynamic objects to leaves within the spatial tree and update their positions as they moves around the world.

We have now discussed all the changes to CBaseLeaf that have been added to facilitate the needs of the rendering system. Most of the methods that we have added are utility methods that will be used by the tree's BuildRenderData method to add and retrieve RenderData structures and Element structures to the leaves as the data is being added to leaf bins.

15.3.3 Leaf Bins

When we cover the rendering enhancements to CBaseTree in a moment, we shall see that one of its new members is an array of CLeafBin structures. A leaf bin is an object that stores the indices of all triangles in the tree that use a specific subset. If multiple vertex buffers are being used by the tree to contain the entire scene, and the subset for a specific leaf bin has geometry spread over those multiple buffers, the leaf bin will contain an index buffer for each vertex buffer being used by that subset. The number of index buffers being used is not necessarily equal to the number of vertex buffers being used by the tree. If the tree's static geometry is contained in five vertex buffers, but subset 6 (for example) had triangles contained in only two of those buffers, the leaf bin for subset 6 will contain only two index buffers. Although the leaf bin need not necessarily use a separate index buffer for each vertex buffer containing triangles of its associated subset, it makes the system cleaner and easier to understand. We know for example that if a leaf bin has triangles in vertex buffers 1 and 3, when the leaf bin is rendered, we will need to set vertex buffer 1 first and its associated index buffer and execute the render batches stored for that buffer combination. Then we will then have to set the second index/vertex buffer pair and execute the render batches relative to that buffer set.

In our earlier discussions, we also discovered that a leaf bin is not just a place where all the static index buffers used by a given subset of polygons reside; it also has methods and structures used during the visibility pass to construct a list of render batches for each buffer combination. These render batches describe which triangles in that leaf bin are currently considered visible by the system and should be rendered when the application wishes to render that subset of the tree.

Each render batch structure represents a block of continuous visible triangles in one of the leaf bin's index buffers, and as such describes a block of triangles in that leaf bin that can be rendered with a single draw call. Because the currently visible triangles stored in a leaf bin may not all be arranged in consecutive order in the leaf bin, many render batches will typically be created representing all the visible sections of the index buffer that need to be rendered. Furthermore, because a leaf bin may have to manage multiple index buffers due to the fact that its triangle data is spread over multiple vertex buffers, a render batch list will need to be constructed for each buffer combination. The render batch lists are destroyed and rebuilt for each leaf bin every time a visibility pass of the tree is performed. We can just

think of a render batch list as simply being a list of drawing instructions for a given buffer pair inside the leaf bin.

CBaseTree will now manage a list (actually an STL map, but more on this in a moment) of leaf bins. The number of leaf bins allocated for the tree will be equal to the number of subsets used by the static polygon data stored in the tree. That is, if the polygon data in the tree belongs to a combination of 15 subsets, the BuildRenderData method will create 15 leaf bins, one for each subset. The BuildRenderData method will then proceed to fill those leaf bins with the indices of the triangles that belong in each leaf bin.

We have already seen the CLeafBin::AddVisibleData method being used by the CBaseLeaf::SetVisible method and we will see many of its other methods being used later during the building of the render data. For now though, we will examine the CLeafBin object, look at the structures and support objects it uses, the methods it exposes and the code to those methods.

15.3.4 The RenderBatch Structure

During the visibility pass, the leaf bin will build a list of RenderBatch structures for each buffer combination used by the leaf bin's triangles. Each RenderBatch structure simply describes a block of triangles in one of the leaf bin's index buffers that can be rendered with a single draw call.

```
struct RenderBatch
{
    unsigned long IndexStart;
    unsigned long PrimitiveCount;
};
```

There will be a RenderBatch list constructed during the visibility pass for each index buffer in use by the leaf bin.

unsigned long IndexStart
This value will contain the position in the associated index buffer of the first index in this run of visible triangles.

unsigned long PrimtiveCount
This describes how many triangles starting at IndexStart are contained in this visible continuous section of the index buffer.

15.3.5 CLeafBinData – The Source Code

A CLeafBin object contains an array of CLeafBinData structures. Each CLeafBinData structure houses a vertex/index buffer combination, information about those buffers, and the RenderBatch list for that vertex and index buffer pair. Rendering a leaf bin involves looping through its CLeafBinData array and setting the vertex and index buffers stored in the current CLeafBinData element being processed before looping through the CLeafBinData's render batch list calling DrawIndexedPrimitive for each one.

If the spatial tree is using five vertex buffers, every leaf bin will contain an array of five CLeafBinData objects as this is the maximum number of vertex/index buffers that could ever be needed by a leaf bin. However, not all the elements in the CLeafBinData array are necessarily used by the leaf bin. If the leaf bin represents geometry that is spread over only two of the five vertex buffers in use by the tree, that leaf bin will only use the first two elements in its CLeafBinData array. Each element in that array would contain the index buffer and vertex buffer to set on the device in order to render the batches for that buffer combination.

The CLeafBinData class is really just a structure with a constructor that initializes its members to zero. Although it has a fair number of members, they really just describe a vertex buffer and an index buffer and information about those buffers (e.g., the amount of data that is contained in them).

This object is declared in CBaseTree.h and is shown below followed by a description of its members.

```
class CLeafBinData
{
public:

    // Constructors & Destructors for This Class.
    virtual ~CLeafBinData( );
             CLeafBinData( );

    // Public Variables for This Class.
    LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer;
    LPDIRECT3DINDEXBUFFER9  m_pIndexBuffer;

    unsigned long           m_nFaceCount;
    unsigned long           m_nVertexCount;
    unsigned char           m_nVBIndex;
    bool                    m_b32BitIndices;
    unsigned long           m_nLastIndexStart;
    unsigned long           m_nLastPrimitiveCount;
    RenderBatch            *m_pRenderBatches;
    unsigned long           m_nBatchCount;
};
```

LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer
This member stores a pointer to the tree's vertex buffer for which this object is applicable. The tree is responsible for allocating and releasing this vertex buffer; this is just a pointer to it so that we can easily

bind it to the device before rendering the RenderBatch instructions stored here. The vertex buffer is not specific to any one leaf bin and may be shared by many leaf bins. All leaf bins that have geometry in one of the tree vertex buffers will all have a CLeafBinData structure that has a pointer to that buffer.

LPDIRECT3DINDEXBUFFER9 m_pIndexBuffer
This is a pointer to an index buffer that contains all the triangles in the above vertex buffer that use the subset associated with the parent leaf bin. This index buffer is not shared by any other leaf bins or even any other CLeafBinData structures within the same leaf bin. This is a static index buffer that is built at leaf bin creation time and it contains all geometry (not just visible geometry) that uses the applicable subset of the leaf bin.

unsigned long m_nFaceCount
Describes how many triangles are in the above index buffer.

unsigned long m_nVertexCount
Describes how many vertices are contained in the vertex buffer pointed to by the first member of this structure.

unsigned char m_nVBIndex
This member describes the vertex buffer pointed to by the first member of this structure as a numerical index in the spatial tree's vertex buffer array. For example, if the spatial tree has stored its geometry in five vertex buffers, but this CLeafBinData structure was representing triangle data stored in the second vertex buffer, this value would be set to 1 (the index of the second vertex buffer in the spatial tree's vertex buffer array).

bool m_b32BitIndices
This member is set to true or false when building the leaf bins to determine whether the index buffer stored here uses 32-bit indices (rather than 16-bit indices). This is certainly useful information to have if we ever wish to lock the index buffer and access the data.

unsigned long m_nLastIndexStart
We saw this member being used when we looked at a simplified version of the leaf bin's AddVisibleData method. It records the first index in a batch of adjacent triangles currently being collected. This value is first set when the AddVisibleData method is called and must start collecting for a new render batch. Every time the function is called after that, if the passed run of adjacent triangles follows on from the ones added to the batch in the previous function call, this value remains unaltered. When we do finally get passed some render data which does not form a continuous block with all the triangles we have collected so far (starting at m_nLastIndexStart), we have collected all we can for that batch and the triangles are committed to a new RenderBatch structure. m_nLastIndexStart is then set to the location of the new run of triangles we are starting to collect.

In short, this value is used as temporary storage during the visibility pass by the leaf bin to record the start of the current batch of adjacent indices collected so far.

unsigned long m_nLastPrimitiveCount
This member is also used as temporary storage during the visibility pass by the CLeafBin::AddVisibleData method. It is used to record, with each function call, the number of adjacent triangles we have collected so far for the current batch being compiled. We discussed this process in some detail earlier when we examined the construction of the RenderBatch lists.

RenderBatch *m_pRenderBatches
This member is an array of RenderBatch structures that will be flushed and filled every time a visibility pass is executed on the tree. This array is allocated (when the CLeafBinData item is first created) to be large enough to hold the maximum number of render batches that we could ever need to create. Although this admittedly wastes some memory, it does allow us to forego any array resizing overhead when building and storing the render batches. As this should be a critically fast process, we want to do everything we can to eliminate such overhead.

This array describes a list of draw instructions for the vertex and index buffers referenced by this structure. Each element in this array represents one DrawIndexedPrimitive call using the above vertex and index buffers.

unsigned long m_nBatchCount
The member stores the number of *valid* RenderBatch structures contained in the above array. Remember, the size of the above array never changes, as discussed above. It is allocated once when the leaf bin first creates this CLeafBinData item and is made large enough to contain the maximum number of render batches that could possibly be created (more on this in a moment). Therefore, this member really describes the number of elements in the above array that are currently considered valid with respect to the last visibility pass. Whenever a new visibility pass is performed, this member is initially set to zero and then incremented every time we add new RenderBatch information. Therefore, this value describes not only the number of batches in the above array that will need to be executed to draw all the visible data in the associated index buffer, but it will also describe the number of DrawIndexedPrimitive calls that will need to be performed to render that data.

15.3.6 CLeafBin – The Source Code

The object that binds this whole system together is CLeafBin. This class has very few member variables as it is really just a wrapper around an array of CLeafBinData structures for a given subset ID. It also has methods that allow you to retrieve the subset/attribute ID of the leaf bin, add and retrieve CLeafBinData structures from its array, and a method that renders the bin. The leaf bin also has the now familiar AddVisibleData method that is called by the CBaseLeaf::SetVisible method to add a leaf's polygon contribution to a render batch. The leaf bins themselves will be created by CBaseTree during the BuildRenderData method.

Below we show the class declaration for CLeafBin which is contained in CBaseTree.h. This is followed by an explanation of its member variables.

```
class CLeafBin
{
public:

    // Constructors & Destructors for This Class.
    virtual ~CLeafBin();
    CLeafBin( unsigned long nAttribID, unsigned long nLeafCount );

    // Public Functions for This Class.
    unsigned long   GetAttributeID      ( ) const;
    CLeafBinData  * GetBinData          ( unsigned long Index ) const;
    bool            AddLeafBinData      ( CLeafBinData * pData );
    void            AddVisibleData      ( unsigned char VBIndex,
                                          unsigned long IndexStart,
                                          unsigned long PrimitiveCount );
    void            Render              ( LPDIRECT3DDEVICE9 pDevice );

private:

    // Private Variables for This Class.
    unsigned long  m_nAttribID;     // The attribute ID of the faces stored here
    unsigned long  m_nLeafCount;    // Total number of leaves in the tree
    CLeafBinData **m_ppBinData;     // An array containing all the information
                                    // we need to render all the data in this bin
                                    // for each vertex bubber (one per primary VB).
    unsigned char  m_nVBCount;      // Number of vertex buffers in use by system
};
```

unsigned long m_nAttribID
This value of this member will be passed into the constructor when the leaf bin is first created by the tree. It contains the subset ID assigned to this leaf bin. That is, all the triangles in the tree that have a matching attribute ID will have their indices stored in this leaf bin.

unsigned long m_nLeafCount
The value of this member is also passed into the constructor when the leaf bin is first created by the tree. It contains the number of leaves in the tree. You will see why we need this value in a moment when we look at the remaining code.

CLeafBinData **m_ppBinData
This is a pointer to an array of CLeafBinData pointers. The size of this array will be equal to the number of vertex buffers being used by the tree. That is, it will hold enough pointers in each leaf bin for the possibility that the leaf bin might have triangles contained in every one of the vertex buffers and would therefore need to represent these triangles with a CLeafBinData structure for each.

Of course, this does not mean that we always allocate a CLeafBinData object for every vertex buffer for each leaf bin, since a given leaf bin might only use two of the vertex buffers and would therefore only need to have two CLeafBinData structures allocated for it. If a leaf bin does not have geometry in one or more of the tree's vertex buffers, the elements in this array that correspond to that vertex buffer will simply be set to NULL. For example, if the tree uses five vertex buffers to store its geometry, the m_ppBinData array in each leaf bin would be allocated large enough to store five pointers. However, if

a leaf bin only has geometry in the first and third vertex buffer, it would have only two CLeafBinData objects allocated for it and have their pointers stored in elements [0] and [2] in this array. The rest of the elements would be set to NULL for the other vertex buffers.

unsigned char m_nVBCount
This value describes the number of vertex buffers in use by the parent tree. This is useful because it also describes the size of the above array. We need this value so that when we render the leaf bin, we know how many element we have to loop through in the above array testing for valid pointers.

Let us now take a look at the methods for this class, which are for the most part, all rather small and straightforward. We will start with the constructor.

Constructor - CLeafBin

The constructor is called by the tree during the generation of its renderable data. The application should never want to allocate a leaf bin. The constructor takes two parameters; the attribute ID of the subset this leaf bin will contain indices for, and the total leaf count of the tree for which the leaf bin is being created. These are stored in the corresponding member variables.

```
CLeafBin::CLeafBin( unsigned long nAttribID, unsigned long LeafCount )
{
    // Store passed values
    m_nAttribID  = nAttribID;
    m_nLeafCount = LeafCount;
    m_ppBinData  = NULL;
    m_nVBCount   = 0;
}
```

Notice that when the leaf bin is first created, the CLeafBinData pointer array is set to NULL and the vertex buffer count member is set to zero. The CLeafBinData structures will be added to the leaf bin (via the CLeafBin::AddLeafBinData method) once the leaf bin has been constructed.

GetAttributeID – CLeafBin

This is a simple accessor function that allows the caller to query the attribute ID of the leaf bin to determine which subset of polygons it contains.

```
unsigned long CLeafBin::GetAttributeID( ) const
{
    return m_nAttribID;
}
```

AddLeafBinData - CLeafBin

This method is called by the BuildRenderData method when it wishes to add a CLeafBinData object to the leaf bin's internal array. You will see in a moment when we discuss the CBaseTree::BuildRenderData method that it essentially loops through each leaf and then through each polygon in those leaves to build an array of vertices and an array of indices into those vertices for each leaf bin. Once the number of vertices we have collected exceeds the number supported in the single vertex buffer by the hardware, we will create the vertex buffer and fill it with the vertices we have collected so far. We will then create index buffers for each leaf bin that index into that vertex buffer. It is at this point that we will inform each leaf bin that we would like to add a new CLeafBinData item to its array which we will populate with the new vertex buffer and index buffer pointers.

The method itself is fairly routine. We pass in a pointer to a pre-allocated CLeafBinData object and the function will resize the leaf bin's array to make room for this pointer at the end. The first part of the function is like every other array resizing method we have discussed, so this should need no explanation.

```
bool CLeafBin::AddLeafBinData( CLeafBinData * pData )
{
    CLeafBinData ** ppBuffer;

    // First allocate space for all the data items
    ppBuffer = new CLeafBinData*[ m_nVBCount + 1 ];
    if ( !ppBuffer ) return false;

    // Any old data?
    if ( m_nVBCount > 0 )
    {
        // Copy over the old data, and release the old array
        memcpy( ppBuffer, m_ppBinData, m_nVBCount * sizeof(CLeafBinData*) );
        delete []m_ppBinData;

    } // End if any old data

    // Set the new entry
    ppBuffer[ m_nVBCount ] = pData;

    // Store the new buffer pointer
    m_ppBinData = ppBuffer;
    m_nVBCount++;
```

At this point we have resized the array and added the passed CLeafBinData pointer to the end of the array. However, what this function will also be responsible for is allocating the passed CLeafBinData structure's RenderBatch array. You will recall from our coverage of the CLeafBinData structure that the RenderBatch array will be allocated just once when the CLeafBinData structure is first added to the leaf bin. It will be made large enough to store the maximum possible number of render batches that could ever be generated for a leaf bin. This way, we will not have to suffer array resizes and flushes as the number of visible leaves fluctuates with each visibility update.

Because the indices of our triangles will be added to the leaf bin index buffers in tree traversal order to help minimize the number of render batches needed, the worst case is that our queue will contain batches where one leaf is visible and the next is invisible, repeating over the entire set. In such a scenario we would need to create a RenderBatch structure for every visible leaf, which in this scenario would half of them. This means that the highest number of batches we will ever need to store in a leaf bin is simply the total number of leaves divided by two. However, as we are performing a division using integer math, we will add one to the total leaf count prior to performing the division. This is to ensure that the result is rounding up and not down. We want 5/3=1.666 to be rounded up to 2 not down to 1. This will make sure the render batch array in each CLeafBinData structure is large enough to cope with the worst case visibility scenario.

Here is the remaining code that allocates the RenderBatch array for the passed CLeafBinData structure before returning.

```
    // Note: We add '1' to ensure that we don't have any rounding issues.
    pData->m_pRenderBatches = new CLeafBinData::RenderBatch[(m_nLeafCount+1) / 2 ];

    if ( !pData->m_pRenderBatches ) return false;

    // Success!
    return true;
}
```

GetBinData - CLeafBin

This method is a simple function that allows the caller to retrieve one of the leaf bin's CLeafBinData pointers. The caller simply passes the index of the element in the array it would like to retrieve the CLeafBinData item for. This may be NULL if the leaf bin does not contain data for the tree's vertex buffer with the same index.

This function essentially allows the tree to say to the leaf bin, "Give me your index data and render batches for a specific vertex buffer index".

```
CLeafBinData * CLeafBin::GetBinData( unsigned long nIndex ) const
{
    // Validate parameters
    if ( nIndex >= m_nVBCount ) return NULL;

    // Retrieve the item
    return m_ppBinData[ nIndex ];
}
```

As you can see, as long as the passed index is within the bounds of the array (i.e., less than m_nVBCount), the pointer contained in the passed index of the CLeafBinData array is returned.

AddVisibleData – CLeafBin

This method should be familiar since we looked at various versions of this type of function as a means to initially understand the render batch system.

We saw earlier that it is called from the CBaseLeaf::SetVisible method. When a leaf is deemed to be visible during the update pass, a loop is made through the RenderData items stored in that leaf (one exists for each subset in the leaf). An inner loop was then used to iterate through the (possible) multiple Elements stored in that RenderData item. There was an Element in RenderData structure for each vertex buffer that contains geometry from that leaf for the given subset. Each Element describes a run of triangles in that leaf that use the subset along with the index of the vertex buffer in which the run of triangles is contained. The AddVisibleData method of CLeafBin is then called and passed the Element information: the vertex buffer index in which the run is stored, the start index in the leaf bin's index buffer associated with that vertex buffer, and the number of triangles in the run.

```
void CLeafBin::AddVisibleData( unsigned char VBIndex,
                               unsigned long IndexStart,
                               unsigned long PrimitiveCount )
{
```

The vertex buffer index of the render element information passed in also describes the index of the CLeafBinData structure in the leaf bin's array of the index buffer and render batch list associated with this vertex buffer. Therefore, we fetch the pointer to the relevant CLeafBinData structure from the bin's array.

```
    CLeafBinData * pData = m_ppBinData[ VBIndex ];
    if ( !pData ) return;
```

We then fetch the values of the CLeafBinData item's m_nLastIndexStart and m_nLastPrimitiveCount members into local variables. m_nLastIndexStart will currently contain the first index of the first triangle in the current render batch we are trying to build for this index buffer. The m_nLastPrimitiveCount member will contain the number of adjacent triangles we have been able to pack into the batch so far. Hopefully, the range of triangles passed in will follow on exactly where m_nLastPrimitiveCount ends in the index buffer, allowing us to add the new triangle range to the current batch being compiled.

```
    ULONG LastIndexStart      = pData->m_nLastIndexStart,
          LastPrimitiveCount  = pData->m_nLastPrimitiveCount;
```

If LastPrimitiveCount equals zero, it means that we have not yet collected any triangles for the current batch and therefore the triangle range passed in should be used to start the beginning of a new render batch. When this is the case, we simply copy the passed IndexStart and PrimitiveCount into the LastIndexStart and LastPrimitiveCount variables. Hopefully, the next time this function is called for another visible leaf, we can add the passed triangle range to the end of this batch we have just started.

```
    // Build up batch lists
```

```
    if ( LastPrimitiveCount == 0 )
    {
        // We don't have any data yet so just store initial values
        LastIndexStart     = IndexStart;
        LastPrimitiveCount = PrimitiveCount;

    } // End if no data
```

If this is not the case then we must be currently in the middle of compiling a batch of adjacent triangles for the CLeafBinData's index buffer. Therefore, we test to see if the start index passed into the function follows on immediately after the current batch we are compiling ends (LastIndexStart+(LastPrimitiveCount*3)). If it does, then the passed triangles continue the adjacent run of triangles we have already recorded and we can just add them to the current batch by increasing the primitive count to compensate for the number of primitives passed into the function.

```
    else if ( IndexStart == (LastIndexStart + (LastPrimitiveCount * 3)) )
    {
        // Just grow the primitive count
        LastPrimitiveCount += PrimitiveCount;

    } // End if consecutive primitives
```

If none of the above cases are true then it means that we are in the middle of compiling a render batch but the triangle(s) we have just been passed do not follow on continuously from the current batch we are compiling. This essentially ends the current batch we are compiling and the new triangles we have just been passed will be used to start a new batch collection process.

```
   else
   {
   // Store any previous data for rendering
   pData->m_pRenderBatches[pData->m_nBatchCount].IndexStart= LastIndexStart;
   pData->m_pRenderBatches[pData->m_nBatchCount].PrimitiveCount=LastPrimitiveCount;
   pData->m_nBatchCount++;

   // Start the new list
   LastIndexStart     = IndexStart;
   LastPrimitiveCount = PrimitiveCount;

   } // End if new batch
```

As you can see, because LastIndexStart and LastPrimitiveCount describe the block of adjacent triangles we managed to collect until this function was called, we store them in the CLeafBinData object's render batch array and increase its render batch count. This render batch is now finished and we need to start a new collection process. We do this by assigning the start index and primitive count of the range of triangles passed in to the LastIndexStart and LastPrimitiveCount local variables which will be used from this point on. Finally we store the local LastIndexStart and LastPrimitiveCount values in the CLeafBinData member variables so that they are available the next time this method is called.

```
    // Store the updated values
    pData->m_nLastIndexStart     = LastIndexStart;
```

```
    pData->m_nLastPrimitiveCount = LastPrimitiveCount;
}
```

Render - CLeafBin

The CLeafBin::Render method is called by the tree to render all the triangles contained in the leaf bin. This method will not typically be called by the application, as it will use the CBaseTree::DrawSubset method for each subset that it wishes to render instead. DrawSubset simply issues the render call to the leaf bin with the matching attribute.

The Render function is shown below. It essentially loops through each pointer in the leaf bin's CLeafBinData array to process and render the data contained in each.

```
void CLeafBin::Render( LPDIRECT3DDEVICE9 pDevice )
{
    ULONG i, j;

    // Loop through each vertex buffer
    for ( i = 0; i < m_nVBCount; ++i )
    {
        CLeafBinData * pData = m_ppBinData[ i ];
        if ( !pData ) continue;
```

As you can see in the above code, the current iteration of the loop is skipped if this leaf bin does not have a CLeafBinData item for a given vertex buffer.

The next function call requires some explanation. You will recall that in the AddVisibleData method we used the m_nLastIndexStart and m_nLastPrimitiveCount members of the CLeafBinData item to record the start and triangle count of the batch of triangles being compiled. Only when we are passed triangles that do not continue the current batch is the batch data copied into a RenderBatch data structure. This means that at the end of the visibility process, we may be in the middle of compiling a batch of triangles that have not yet been committed to the render batch list. For example, consider the last leaf processed for a given subset. Even if it starts a branch new batch, because no further triangles are passed, the AddVisibleData function is never passed triangles that break the batch, and therefore it is never aware that the batch ever ended. This means that the m_nLastIndexStart and m_nLastPrimitiveCount members of each CLeafBinData structure describe their final batches, but they are not yet committed to a render batch structure. Therefore, before we draw the CLeafBinData render batches, we must call AddVisibleData for this item one last time, passing in both an index start and primitive count of zero.

```
        // Commit any last piece of data to the render queue and reset "Last"
        // values back to 0
        AddVisibleData( (unsigned char)i, 0, 0 );
```

As the first parameter, we pass the current vertex buffer index we are processing which describes (to the AddVisibleData method) which CLeafBinData object we are adding triangles for. Also, because we pass 0 in as the IndexStart parameter, which could not possibly continue any batch that has been compiled,

the function will recognize that the last batch has ended and will store it in the CLeafBinData object's render batch list. Finally, this will cause the function to start a new batch to be compiled with an index start and primitive count set to the values passed in. Since we pass zero for both of these values, this essentially resets the CLeafBinData's m_nLastIndexStart and m_nLastPrimitiveCount members back to zero for the next visibility pass.

At this point, we have the CLeafBinData pointer, so we send its vertex and index buffers to the device and set the FVF for the vertex type being used.

```
        // Set the stream sources to the device
        pDevice->SetStreamSource( 0, pData->m_pVertexBuffer, 0, sizeof(CVertex) );
        pDevice->SetIndices( pData->m_pIndexBuffer );

        // Set the FVF
        pDevice->SetFVF( VERTEX_FVF );
```

Finally, we loop through every RenderBatch structure stored in the CLeafBinData's render batch list and render the triangles described by each.

```
        // Render the leaves
        for ( j = 0; j < pData->m_nBatchCount; ++j )
        {
            CLeafBinData::RenderBatch & Batch = pData->m_pRenderBatches[j];

            // Render any data
            pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                           0,
                                           0,
                                           pData->m_nVertexCount,
                                           Batch.IndexStart,
                                           Batch.PrimitiveCount );
        } // Next element
    } // Next Vertex Buffer
}
```

We have now covered all of the rendering code for CBaseLeaf, CLeafBin, and CLeafBinData and you should have a fairly good understanding of how these components work. We will now cover the new rendering code in the top level object (CBaseTree) that brings all of these components together under one system.

15.4 CBaseTree – Adding Rendering Support

Adding the code that actually performs the visibility pass and renders the data in the tree is relatively trivial. The CBaseTree methods that perform these tasks will be small and simple as they just pass the requests on to the leaf bins. As we have seen, it is the leaf bins that contain the triangles and have knowledge of how to render them.

The majority of work we will have to do in CBaseTree is the implementation of the BuildRenderData method. As you will soon see, this method is responsible for building all the vertex buffers used by the tree, allocating the leaf bins for each attribute used by the tree, building the index buffers for each leaf bin, and a host of other tasks. Most of our discussion of CBaseTree will be spent describing how this function works. If you have already looked at the source code to this function, you might have noted that it seems a little intimidating. Do not worry though; we will break this function down (and the functions it calls) piece by piece so that we understand everything that is going on when building the render data.

15.4.1 CBaseTree – The Source Code

Surprisingly few methods and members have to be added to CBaseTree in order to implement the rendering system. The upgraded class declaration is shown below. The new methods and members that pertain to the rendering system are highlighted in bold. This declaration will be followed by a detailed description of the new members and an examination of each new method.

```
class CBaseTree : public ISpatialTree
{
public:

    // Friend list for CBaseTree
    friend void CBaseLeaf::SetVisible( bool bVisible );

    // Constructors & Destructors for This Class.
    virtual ~CBaseTree( );
             CBaseTree( LPDIRECT3DDEVICE9 pDevice, bool bHardwareTnL );


    // Public Virtual Functions for This Class (from base).
    virtual bool            AddPolygon        ( CPolygon * pPolygon );
    virtual bool            AddDetailArea     ( const TreeDetailArea &DetailArea );

    virtual bool            Repair              ( );

    virtual PolygonList    &GetPolygonList      ( );
    virtual DetailAreaList &GetDetailAreaList   ( );
    virtual LeafList       &GetLeafList         ( );
    virtual LeafList       &GetVisibleLeafList  ( );

    virtual void            DrawSubset          ( unsigned long nAttribID );
    virtual void            ProcessVisibility   ( CCamera & Camera );


    // Public Functions for This Class
    CLeafBin *              GetLeafBin          ( unsigned long nAttribID );
    bool                    AddLeaf             ( CBaseLeaf * pLeaf );
    bool                    AddVisibleLeaf      ( CBaseLeaf * pLeaf );
    bool                    BuildRenderData     ( );

protected:
```

```
    // STL Typedefs
    typedef std::map<ULONG,CLeafBin*>       LeafBinMap;

    // Protected Functions for This Class.
public:
    void   DrawBoundingBox     ( const D3DXVECTOR3 & Min,
                                 const D3DXVECTOR3 & Max,
                                       ULONG Color,
                                       bool bZEnable = false );

protected:
    void   DrawScreenTint      ( ULONG Color );
    void                    CalculatePolyBounds ( );

    void                    RepairTJunctions    ( CPolygon * pPoly1,
                                                  CPolygon * pPoly2 );

    ULONG                   WeldBuffers ( ULONG VertexCount,
                                          CVertex * pVertices,
                                          std::map<ULONG,ULONG*> & BinIndexData,
                                          std::map<ULONG,ULONG> & BinSizes,
                                          ULONG * pFirstVertex = NULL,
                                          ULONG * pPreviousVertex = NULL );

    bool                    CommitBuffers( ULONG VertexCount,
                                           CVertex * pVertices,
                                           std::map<ULONG,ULONG*> & BinIndexData,
                                           std::map<ULONG,ULONG> & BinSizes,
                                           bool b32BitIndices );

    bool                    PostBuild           (  );


    // Protected Variables for This Class.
    LPDIRECT3DDEVICE9           m_pD3DDevice;
    LPDIRECT3DVERTEXBUFFER9    *m_ppVertexBuffer;
    unsigned char               m_nVBCount;
    bool                        m_bHardwareTnL;

    LeafBinMap                  m_LeafBins;

    LeafList                    m_Leaves;
    PolygonList                 m_Polygons;
    DetailAreaList              m_DetailAreas;

    LeafList                    m_VisibleLeaves;
};
```

Let us discuss a new type definition we have in this namespace:

```
// STL Typedefs
   typedef std::map<ULONG,CLeafBin*>  LeafBinMap;
```

We will store the tree's leaf bins in an STL map. You will recall from our C++ programming course (Module II) that an STL map is basically a hash table that allows us to store key/value pairs in a manner that makes it very efficient to look up that value by specifying the key. We type define LeafBinMap to be an STL map that stores a ULONG as the key and a CLeafBin pointer as the value paired with that key. The key will by the attribute ID of the leaf bin pointed to by the value assigned to that key. Therefore, the tree would be able to very efficiently fetch the leaf bin for subset 6's triangles (for example) by doing something like this:

```
CLeafBin = LeafBinMap [ 6 ];
```

This looks like a simple array look up, so why do we not just use an array to store the leaf bin pointers where the attribute ID is the index into the array where the associated leaf bin is stored?

What we must remember is that our scene uses global attribute IDs and that the static geometry compiled in the tree may only use some of these. For example, assume that after loading our scene and all of our dynamic objects that the scene has created 120 attributes. Imagine however, that the static polygons in the spatial tree only use subsets 6, 40, and 120 out of the 120 possible scene attributes. The leaf bin array would need to be 120 elements in size when only 3 of the elements would contain valid pointers to leaf bins. That is a waste of memory. Alternatively, the map would only have three rows in its hash table. Of course, you could use an array that is the exact size of the number of attributes used by the tree but that would mean the array indices no longer match the attribute IDs of the leaf bins stored there. In this case, whenever you need to fetch the leaf bin pointer for a given subset, you would have to loop through the array searching for the leaf bin with the attribute ID you are looking for. This will hamper performance when the tree contains a lot of attributes (which would have to be searched through every time the DrawSubset method is called). This is especially true if the tree data requires multiple render passes. A map is definitely a better choice of data structure here since it conserves memory and performs efficient value lookups.

Let is now take a look at the new member variables in CBaseTree that have been added to support the render system.

LPDIRECT3DDEVICE9 m_pD3DDevice
This pointer is passed into the constructor of CBaseTree (called by the constructor of the derived classes) and represents the device which the tree will use to render. If this is set to NULL, the BuildRenderData method will perform no action and will return immediately. This is very useful if you intend to use the tree only for collision queries. Passing NULL in as this parameter to the constructor of a derived class allows you to prevent the render system from being activated and avoid the unnecessary memory consumption. No vertex buffers, index buffers, or leaf bins will be generated in this instance. By setting this to point at a valid device, we inform CBaseTree that we intend to use its render system.

LPDIRECT3DVERTEXBUFFER9 *m_ppVertexBuffer
This will be set to NULL until CBaseTree::BuildRenderData executes. At the end of the renderable data building process is will point to an array of vertex buffer pointers. That is, if it was determined during the building of the render data that the static scene needs to be contained in three vertex buffers, this will point to an array of three vertex buffer pointers currently containing the vertex data of the tree's static geometry.

unsigned char m_nVBCount
After the render data building process has been completed, this member will contain the number of vertex buffers being used by the render system and the number of elements in the above vertex buffer pointer array.

bool m_bHardwareTnL
This Boolean is also passed into the constructor of CBaseTree (called from the derived classes) and tells the render system whether the graphics hardware on the current system on which the application is running supports transformation and lighting of vertices in hardware. This Boolean will ultimately determine the vertex buffer creation flags that need to be passed to DirectX when the tree needs to create a new vertex buffer.

LeafBinMap m_LeafBins
This is an STL map containing the leaf bin pointers used by the tree. This map will be empty until CBaseTree::BuildRenderData is called. As the leaf bins are created, their pointers are added to this map. The key for each row in the table is the attribute ID of the leaf bin. The value assigned to that key will be the leaf bin pointer.

LeafList m_VisibleLeaves
We discussed this new leaf list when we discussed the CBaseLeaf source code. This leaf list is emptied just before a new visibility determination pass is performed on the tree. As each visible leaf is encountered, its pointer is added to this list. We saw the code that added the leaf pointers to this list in the CBaseLeaf::SetVisible method earlier in the lesson. At the end of each ISpatialTree::ProcessVisibility call, this list will contain the pointers for all the leaves that are currently deemed visible by the render system (i.e., leaves that are currently inside the camera frustum).

We will now discuss the new methods in CBaseTree that pertain to the rendering system.

AddVisibleLeaf – CBaseTree

We saw this method being called earlier in the lesson when we examined the code to CBaseLeaf::SetVisible. It is passed the pointer of a leaf which it then adds to the tree's visible leaf list. Remembering that LeafList is just an STL list, we can see that this function simply adds the passed pointer to the end of the list.

```
bool CBaseTree::AddVisibleLeaf( CBaseLeaf * pLeaf )
{
    try
    {
        // Add to the leaf list
        m_VisibleLeaves.push_back( pLeaf );

    } // End Try Block

    catch ( ... )
```

```
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

The application should never call this function; it should be left to the tree's visibility system to build this list with each visibility pass. In particular, it is only the CBaseLeaf::SetVisible method that ever calls this function to add its own pointer to the list.

GetVisibleLeafList - CBaseTree

This method is exposed purely for the application's benefit. The application can call this method to get returned the list of currently visible leaves. This is the leaf list that was compiled the last time the application called the ISpatialTree::ProcessVisibility method.

```
CBaseTree::LeafList & CBaseTree::GetVisibleLeafList()
{
    return m_VisibleLeaves;
}
```

While it might not seem apparent just yet why the application may want to know which leaves are visible (especially when the tree takes care of rendering its visible polygon data), we will see later how this information will be important for the efficient rendering of dynamic objects. The application is going to be responsible for rendering these objects (not the tree) and thus, retrieving the list of currently visible leaves from the tree and then retrieving the dynamic object pointers stored in those leaves, provides the application with the list of currently visible dynamic objects that need to be rendered.

GetLeafBin - CBaseTree

The GetLeafBin method will generally never be called by the application. It is used by the tree during the building of the render data to fetch a leaf bin pointer from the map. For example, if the build process finds a polygon with an attribute of 7 it knows that it has to add its indices to leaf bin 7. As this leaf bin might have already been created (when processing another polygon with the same attribute ID), this method will quickly retrieve the pointer to that leaf bin and return it to the building process. If there is currently no key in the leaf bin map that matches the passed attribute ID the function returns NULL. This informs the calling function that the leaf bin associated with subset 7 has not yet been created, so it should do so now.

```
CLeafBin * CBaseTree::GetLeafBin( unsigned long nAttribID )
{
    LeafBinMap::iterator Item = m_LeafBins.find( nAttribID );
```

```
    // We don't store this attrib ID
    if ( Item == m_LeafBins.end() ) return NULL;

    // Return the actual item
    return Item->second;
}
```

Although it might seem strange that we have not just indexed into the map using the attribute ID (using an array style approach), if we do this and the key/value pair does not exist in the table, this new row will be added with a value of NULL. However, we want this function to return the pointer only if it does exists and not create the row in the hash table if it does not. For this reason, we use the STL map's find method.

The find method does not perform a search through the entire table row by row as its name would perhaps imply. It efficiently jumps straight to the row in the table with the matching key and returns the iterator to that item in the table. If the key (attribute ID) did not exist in the map, then the iterator will point to the end of the map (like a hash table's equivalent of a NULL return). If this is the case, we return NULL from the function. Otherwise we fetch the leaf bin pointer stored in the returned iterator's 'second' member and return it (the key itself is stored in the 'first' member of the iterator).

We have now looked at the small utility functions in CBaseTree which we will need during the render data building process. In the next section, we will discuss the new methods in CBaseTree that are executed once, just after the tree has been built, to compile the tree's static polygon data into vertex and index buffers, and leaf bins.

15.4.2 The Building Phase – Compiling the Render Data

The entire building phase is invoked from the CBaseTree::PostBuild method introduced in the previous lesson. This method is called by the derived classes just after the tree has been built (at the very bottom of their Build functions). We discussed how and why this function calls the CalculatePolyBounds function in the previous lesson to build and store the AABBs for each CPolygon in the tree. What we have not yet seen are the contents of the second method that it calls. The BuildRenderData method is the method that compiles the data into a renderable format. It creates all the vertex and index buffers used by the tree and creates and populates the leaf bins.

```
bool CBaseTree::PostBuild(  )
{
    // Calculate the polygon bounding boxes
    CalculatePolyBounds( );

    // Build the render data
    return BuildRenderData( );
}
```

BuildRenderData - CBaseTree

This function is a very large and seemingly complex function. The function could be made shorter by moving multiple processes into a single loop (instead of doing a separate loop for each process), but this would have made the function must harder to break into components for the purpose of explanation. Before we dive into the code, we will discuss what purpose of this function is, the tasks it must perform, and the way those tasks will be implemented. We will then find that this code is not nearly as intimidating as it first seems.

When the function is first called, no vertex buffers or index buffers will have been created. The function will allocate a temporary array of vertices that will be used to collect the vertices of each polygon it processes. We can think of this array initially as collecting the vertices for the first (and possibly only) vertex buffer that will be created. Likewise, the function will also allocate an array of temporary buffers to create the indices for each leaf bin. That is, if the scene contains polygon data for 15 subsets, we will create an array to store 15 sets of indices, one for each leaf bin. We can think of these initial index arrays as being used to collect the indices that are associated with the first vertex buffer.

The thing the function will do is call the CBaseTree::Repair method. We discussed this function in the previous lesson and saw how removes T-junctions in the polygon dataset. We call this automatically in the case of this particular function because we know this data is being used for rendering and we certainly do not want any T-junctions in our renderable data (they cause lighting artifacts and generate sparklies). After the Repair function returns, we have our data ready to be compiled for rendering with all T-junctions removed.

The next step will be to call the CollectLeavesAABB method passing in the bounding box of the root node. This will collect all the leaves of the tree into a leaf list in the exact order in which the tree would be traversed if every leaf was visible. This is very important as it allows us to add the triangles from each leaf into the leaf bin index buffers in the order in which they will be traversed. This greatly increases our changes of creating large render batches during the visibility pass.

Once we have the leaf list, we will loop through and process each leaf. For each leaf we will process the polygons in that leaf. For each polygon we will retrieve its attribute ID and fetch the associated leaf bin for that subset. If a leaf bin has not yet been created for that subset, we will create a new leaf bin for it. The whole point of this process is to make sure that at the end of the loop we have a leaf bin created and stored in the tree for every subset used by the tree. As we process each polygon, we will also calculate the number of indices that should be generated for it and add it to the temporary index array for that leaf bin. Therefore, at the end of the leaf loop we will have made sure that a leaf bin has been created for every subset used by the tree, and we will have recorded exactly how many indices we will need to allocate the temporary index arrays (for each leaf bin) to hold.

Before we go any further, let us cover the first sections of the code that perform these steps. First we will look at the local variables allocated at the top of the function as some of them are extremely significant.

```
bool CBaseTree::BuildRenderData(  )
{
```

```
    LeafList::iterator LeafIterator;
    map<ULONG,ULONG>   BinSizes;
    map<ULONG,ULONG*>  BinIndexData;
    LeafList           Leaves;
    ULONG              nMaxVertexCount, nPolygonCount, nTotalVertices;
    ULONG              nVertexCount, i, j;
    D3DCAPS9           Caps;
    CBaseLeaf        * pLeaf;
    CLeafBin         * pLeafBin;
    CVertex          * pVertices = NULL;
    bool               b32BitIndices = false;
    D3DXVECTOR3        Min, Max;

    CBaseLeaf::RenderData          * pRenderData;
    CBaseLeaf::RenderData::Element * pElement;
```

Notice the two local variables that are highlighted in bold in the above code. These two STL maps are vital to the building process. The first stores ULONGs as both its keys and values and will be used in the initial loop (just described) to record the number of indices we found for each subset/leaf bin. The key of each row in the hash table will be the attribute ID and the value will be the number of indices we recorded that will need to be generated for it. The second STL map is the partner of the previous one and will be used to store the temporary index arrays being collected for each leaf bin. The key is a ULONG which once again stores the attribute ID that its value is relative to. The value is a ULONG pointer that will be allocated to store as many indices as is described by the row in the BinSizes map with the matching key (attribute ID).

For example, imagine that after looping through the leaf list and processing however many indices will be generated by all the polygons, we find that the tree uses subsets 3, 6, and 9. Let us also assume that the number of indices recorded for each subset was 100, 200, and 300, respectively. At the end of the first loop, the following key/value pairs will be stored in the BinSizes map.

BinSizes [3] = 100
BinSizes [6] = 200
BinSizes [9] = 300

The three rows in the map now tell us that we will need to create a temporary index array to collect the 100 indices for the first leaf bin (subset 3), an array to collect 200 indices for the second leaf bin (subset 6) and an array of 300 to collect the indices for the third leaf bin (subset 9).

After the loop, we can then allocate these temporary arrays in the BinIndexData map with code like the following:

```
// Allocate temporary buffers for each of the bin items
map<ULONG,ULONG>::iterator BinSizeIterator = BinSizes.begin();
for ( ; BinSizeIterator != BinSizes.end(); ++BinSizeIterator )
{
    ULONG AttribID = BinSizeIterator->first;
    ULONG Size     = BinSizeIterator->second;

    if ( Size > 0 )
```

```
            BinIndexData[ AttribID ] = new ULONG[ Size ];
        else
            BinIndexData[ AttribID ] = NULL;

        // Reset sizes back to 0, we're going to tally as we go
        BinSizes[ AttribID ] = 0;

    } // Next Bin
```

As you can see, we loop through each row in the BinSizes table (one per subset used by the tree) and extract the key (attribute ID) and the indices size. We then use this size to allocate an array of indices which is assigned to the row in the BinIndexData map with a matching attribute. The function would then proceed to fill up these index arrays with the indices intended for each leaf bin.

Now that we know how these two maps will be used in tandem to represent the index arrays being compiled for each leaf bin, let us examine the first section of the function that creates the leaf bins and records the number of indices that will be needed for each subset.

The first thing we do is test to see if the device pointer is valid. If not, there is no point in initializing the render system and we simply return. Also, if the tree has no leaves we also have no polygon data so we return false. Provided the device and the leaf array are valid, we then call the CBaseTree::Repair method to remove any T-junctions that may exist in the polygon data.

```
    // Instructed not to build data?
    if ( !m_pD3DDevice ) return true;

    // Anything to do ?
    if ( m_Leaves.size() == 0 ) return false;

    // First thing we need to do is repair any T-Junctions created during the build
    Repair( );
```

We will now need to fetch the list of leaves contained in the tree. However, as this leaf list will be iterated through in order and each leaf's polygons added to the leaf bins, we want this leaf list ordered in the same way that the leaves will be encountered when the visibility traversal is performed. This will ensure that we get the most optimal render batches during visibility traversal. To do this we simply fetch the bounding box of the scene (i.e., the root node's bounding box) and feed this box into the CollectLeavesAABB method. When the function returns, the local leaf list passed in as the first parameter will be filled with all the leaves contained in the tree in traversal order.

```
    // Retrieve the leaf list in TRAVERSAL order to ensure that the
    // render batching works as efficiently as possible
    GetSceneBounds( Min, Max );
    CollectLeavesAABB( Leaves, Min, Max );
```

We will now set up a loop that will loop through each leaf and record the total number of indices that will be generated for each subset. We will also allocate the leaf bins for each subset.

For each leaf we fetch its polygon count and then loop through each polygon in the leaf. Notice also that at the top of the next section of code, we set the local variable nTotalVertices to zero. This will be used to also record the total number of vertices stored in the tree (from all polygons).

```
    // Iterate through all of the polygons in the tree and allocate/add to
    // leaf bins.  In addition, total up the index counts required for each bin.
    nTotalVertices = 0;

    for ( LeafIterator = Leaves.begin();
          LeafIterator != Leaves.end();
          ++LeafIterator )
    {
        // Retrieve leaf
        pLeaf = (CBaseLeaf*)(*LeafIterator);
        if ( !pLeaf ) continue;

        // Loop through each of the polygons stored in this leaf
        nPolygonCount = pLeaf->GetPolygonCount();

        for ( i = 0; i < nPolygonCount; ++i )
        {
```

In this loop we fetch a pointer to each polygon in the leaf. If for some reason its vertex count is smaller than 3, then this is an invalid triangle and we skip it. We also skip the polygon if its m_bVisible Boolean is set to false. The application can set a CPolygon's visibility status to false prior to registering it with the tree. This will allow the application to register polygons that will still be used for collision detection queries but will not be rendered. You will see later when we discuss the changes to our application that we set this flag to false for transparent polygons before they are registered with the tree. Transparent polygons will be rendered by our BSP tree (discussed in the next chapter) so that we can get perfect back to front ordering. However, we still want to register such polygons with the spatial tree so that a 'Glass' polygon for example is still considered a solid polygon by the collision system. Therefore, this flag allows our application to say to the tree, "Here is a static polygon for collision purposes, but do not render it as I will render it on my own using a different methodology."

```
            CPolygon * pPoly = pLeaf->GetPolygon( i );

            // Skip if it's invalid
            if ( !pPoly || pPoly->m_nVertexCount < 3 ) continue;

            // Skip if it's not visible
            if ( !pPoly->m_bVisible ) continue;

            // Retrieve the leaf bin for this attribute
            pLeafBin = GetLeafBin( pPoly->m_nAttribID );
```

In the code above, we see that once we have a pointer to a valid renderable CPolygon, we pass its attribute ID into the CBaseTree::GetLeafBin function. If a leaf bin has already been created for this subset, a pointer to the leaf bin will be returned and stored in the pLeafBin local variable pointer. If we have encountered a polygon which belongs to a subset that we have not yet encountered and created a

leaf bin for, the GetLeafBin method will find no such leaf bin and the returned value of NULL will be stored in pLeafBin.

If pLeafBin is NULL then it means we have found a polygon that a leaf bin does not yet exist for, so we had better create one. In the next section of code, we allocate a new leaf bin (passing in the attribute ID the leaf bin is intended to store triangles for and the total leaf size of the tree) and store the leaf bin's pointer in the CBaseTree's m_LeafBins map with a key equal to the attribute ID.

```
            // Determine if there is a matching bin already in existance
            if ( !pLeafBin )
            {
                try
                {
                    // Allocate a new leaf bin
                    pLeafBin = new CLeafBin( pPoly->m_nAttribID, m_Leaves.size() );
                    if ( !pLeafBin ) throw std::bad_alloc(); // VC++ Compat

                    // Add to the leaf bin list
                    m_LeafBins[ pPoly->m_nAttribID ] = pLeafBin;

                } // End Try Block

                catch ( ... )
                {
                    // Clean up and fail
                    if ( pLeafBin ) delete pLeafBin;
                    return false;

                } // End Catch Block

            } // End if no bin existing
```

At this point we have either confirmed that a bin exists that will accept this polygon or we have created a new one that will collect polygons of this type. We will now increment the value stored in the row of the BinSizes map that has a key equal to the current polygon's subset. This allows us to record the indices for each subset in the BinSizes map as we go. We will need to break our clockwise winding polygons into a series of triangles to store in a triangle list (for the index buffer), and we have previously discovered that the number of triangles produced will be equal to the number of vertices in the polygon minus 2 (see Figure 15.3).

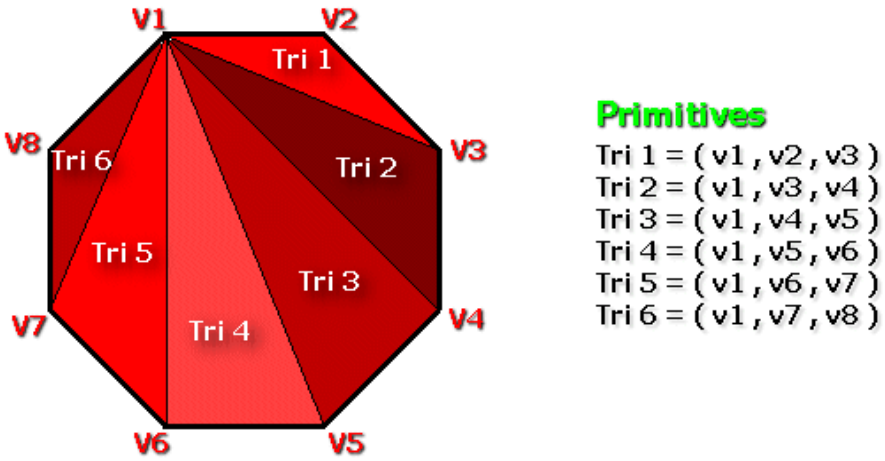


Figure 15.3

In this example we can see that the 8 vertices describe 6 triangles (vertex count - 2). Therefore, as each triangle will be described by three indices in the index buffer, the total number of indices that will be generated by the current polygon will be (VertexCount - 2) * 3.

In the following code you can see that we perform this calculation and add the resulting index count to the value of the row in the map with a key that matches the attribute ID of the polygon. We also add the vertex count of the polygon to the nTotalVertices local variable so that we record the total number of vertices in the entire tree after the entire leaf loop completes.

```
            // Add each of the triangles to the total index count.
            BinSizes[ pPoly->m_nAttribID ] += (pPoly->m_nVertexCount - 2) * 3;

            // Total up vertices
            nTotalVertices += pPoly->m_nVertexCount;

        } // Next Polygon
    } // Next Leaf
```

At this point we are no longer in any loops and we have created the leaf bins for every subset in the tree. We have also recorded the total number of vertices stored in the tree in the nTotalVertices variable. Finally, we have recorded, for each subset, the total number of indices that will be generated for that subset. These index counts are stored in the BinSizes map, keyed by subset/attribute ID. At this point we have collected no vertices or indices and we have not yet added any data to any leaf bins.

What we know is how many vertices we are going to need to store, which allows us to find out two things. First, if the vertex count is larger than the maximum number of vertices the device allows to be stored in one vertex buffer, we know we will need multiple vertex buffers. Second, if the number of vertices we have exceeds 65,535 (0xffff), we know that we will need 32-bit indices to reference them as we cannot fit values greater than this size into a 16-bit index.

First we get the capabilities of the device and retrieve the value of the maximum vertex index capability. This is stored in the D3DCAPS9::MaxVertexIndex member. As the maximum vertex index tells us the maximum index of a vertex that is allowed on the current device, adding one to this amount (remembering that indices are zero based) tells us the total number of vertices that can be placed in a single vertex buffer on this device. We store this value in the nMaxVertexCount local variable.

```
    // Calculate the maximum vertex count the card is capable of storing
    m_pD3DDevice->GetDeviceCaps( &Caps );
    nMaxVertexCount = Caps.MaxVertexIndex + 1;
```

Now that we know the maximum number of vertices that can be stored in a vertex buffer, we should test to see if this is smaller than the total number of vertices used by our tree's static geometry. If it is not, then it means the current device can handle all the vertices we have in a single vertex buffer. When this is the case we simply modify the value of nMaxVertexCount to equal the total number of vertices in the scene. We can think of the nMaxVertexCount member as containing the vertex count at which we will have to create a new vertex buffer during the vertex collection process.

```
    // If it's more than capable, just clamp it to the scene vert count
    if ( nMaxVertexCount > nTotalVertices ) nMaxVertexCount = nTotalVertices;
```

At this point, nMaxVertexCount will contain either the total number of vertices in our tree (if the hardware can handle them all) or the maximum number of vertices that can fit in a single vertex buffer on this device. If this is larger than 65,535 (0xffff) then we will need to create 32-bit index buffers, so we set the local b32BitIndices Boolean to true so we will know this when creating leaf bin index buffers later.

```
    // Determine if we need to use 32 bit indices or not
    b32BitIndices = ((nMaxVertexCount - 1) > 0xFFFF) ? true : false;
```

Since nMaxVertexCount describes the maximum size that the vertex buffers we create will be, we only create a vertex buffer once we have collected this many vertices. Therefore, we create a temporary vertex array of this size that will be used to collect the vertices for each vertex buffer. Once this buffer is full, a hardware vertex buffer will be created, the data copied over, and the array's count set back to zero again. The array will then be reused to collect vertices for the next vertex buffer, and so on.

```
    // Allocate temporary vertex array
    pVertices = new CVertex[nMaxVertexCount];
    if ( !pVertices ) return false;

    nVertexCount = 0;
```

The BinSizes map contains the total number of indices that will be generated for each leaf bin's index buffers. Just as with the temporary vertex array, we will use the BinIndexData map to store pointers to temporary index buffers that will be used to collect the indices for each leaf bin for the current vertex buffer. Once the pVertices array is full and the vertices are committed to a vertex buffer, index buffers will be created for each subset that we collected indices for and the contents of these temporary index arrays will be copied into the index buffers. Each new index buffer will then be added to a leaf bin and

the temporary index arrays stored in the BinIndexData map will be reused to collect the indices for each subset for the next vertex buffer, and so on.

The next section of code loops through every row in the BinSizes map and fetches each key/value pair into the AttribID and Size local variables. If the size of a given row is greater than zero then it means that there are some indices that will be generated for this leaf bin during the index collection process (shown in a moment). Therefore, we allocate an array of indices for each leaf bin used by the scene that is large enough in size to store all the indices recorded for that leaf bin in the above code. After this loop is finished, the LeafIndexData map will store a pointer to an index array for each leaf bin.

```
    // Allocate temporary buffers for each of the bin items
    map<ULONG,ULONG>::iterator BinSizeIterator = BinSizes.begin();
    for ( ; BinSizeIterator != BinSizes.end(); ++BinSizeIterator )
    {
        ULONG AttribID = BinSizeIterator->first;
        ULONG Size     = BinSizeIterator->second;

        if ( Size > 0 )
            BinIndexData[ AttribID ] = new ULONG[ Size ];
        else
            BinIndexData[ AttribID ] = NULL;

        // Reset sizes back to 0, we're going to tally as we go
        BinSizes[ AttribID ] = 0;

    } // Next Bin
```

Notice that the BinSizes map which had its values recorded in the first loop has only been used for the allocation of the temporary index arrays for each subset. As we allocate the index array for each leaf bin, we reset its associated BinSize value to zero. That is because we will use the BinSize map later to record how many indices we have currently added to each leaf bin, which will be used in the generation of the leaf RenderData structures (more on this in a moment).

In the next section of code we will loop through the leaf list again, and through each leaf's polygon list and actually collect the indices and vertices of the polygons into their respective arrays. We will also create and populate the RenderData structures at each leaf (one for each subset contained in that leaf) and create the Elements for each RenderData structure. Remember, each Element in a RenderData structure contains a triangle range (index start and primitive count) for the current vertex buffer being compiled. If all the vertices fit in a single buffer, this loop will only ever create one render element in each leaf's RenderData structure(s). We shall also see two calls to CBaseTree member functions we have not yet covered. The first is called CBaseTree::CommitBuffers and is called once the vertex buffers are filled. It is this function which takes the vertices collected in the temporary vertex array and copies them into a newly allocated vertex buffer. It is also this function that copies the data stored in the index arrays for each subset and copies them into newly allocated index buffers. These index buffers are then stored in newly allocated CLeafBinData structures and added to the leaf bins. We will look at the code to this function in a moment. For now just know that this function is called when we have collected enough vertices and need to create actual Direct3D vertex and index buffers. It is called CommitBuffers because it is this function that takes the data stored in the temporary vertex and index arrays and commits them to static vertex and index buffers. After the CommitBuffers function returns, the data in

the temporary vertex and index arrays are discarded and the current vertex buffer count will have been increased from zero to one. If there are still more leaves or polygons left to the process, the cycle repeats and vertex and index data continues to get collected for the next round of vertex and index buffers that will be created the next time CommitBuffers is called. The code to this function will be covered after we have finished discussing the BuildRenderData method.

The second new CBaseTree method called inside this loop is the WeldBuffers method. This function will be called whenever we fill up the temporary vertex array in an effort to compact the data and get rid of duplicated vertices. We have discussed welding many times throughout this course and understand that it will collapse multiple vertices that share identical properties into a single vertex. This potentially reduces the vertex count and will allow us to fit more vertices in the buffer. Of course, if the vertices are altered or compacted, the weld operation will also need to update the indices so that triangles that indexed vertices that have been removed are now mapped to the vertex they were collapsed onto. Once we find that we cannot add any more polygon vertices to the temporary vertex array without exceeding the maximum vertex count of the hardware, we will call the weld function to try to compact the vertex data in this array and make room for some more vertices. This function will need to be passed not only the temporary vertex array that will be compacted, but also the map of index buffers containing the indices currently collected for each subset that index into this vertex array. These index arrays will also have to be updated so that they correctly index into the compacted vertex array. The contents of this function will also be discussed shortly.

Continuing where we left off, we are now at the point where we have allocated an array to hold the vertices we are going to collect for the vertex buffer we are about to build. We have also allocated an array of indices to record the indices generated for each leaf bin (stored in the BinIndexData map).

We will now loop through each leaf and through each polygon in that leaf. If the current polygon being processed is either invalid or invisible, we skip it and continue to the next iteration of the polygon loop.

```
    // Now we actually build the renderable data for the TRAVERSAL ordered
    // leaf list.
    for ( LeafIterator = Leaves.begin();
          LeafIterator != Leaves.end();
          ++LeafIterator )
    {
        // Retrieve leaf
        pLeaf = (CBaseLeaf*)(*LeafIterator);
        if ( !pLeaf ) continue;

        // Loop through each of the polygons stored in this leaf
        nPolygonCount = pLeaf->GetPolygonCount();

        for ( i = 0; i < nPolygonCount; ++i )
        {
            CPolygon * pPoly = pLeaf->GetPolygon( i );

            // Skip if it's invalid
            if ( !pPoly || pPoly->m_nVertexCount < 3 ) continue;

            // Skip if it's not visible
            if ( !pPoly->m_bVisible ) continue;
```

At this point we have a pointer to the current polygon we are going to process. First we will assign a ULONG pointer to point at the index array in the BinIndexData map for the polygon's subset. We will also assign a pointer to point at the BinSizes value for this subset. This will be set to 0 if this is the first polygon with the current subset we have found so far in the loop.

```
            ULONG * pIndexData = BinIndexData[ pPoly->m_nAttribID ];
            ULONG * pBinSize   = &BinSizes[ pPoly->m_nAttribID ];
```

Just so we understand these variables, pIndexData points to the index buffer we allocated earlier for the leaf bin with the same attribute ID as the current polygon. This pointer will be used to store the indices of the polygon in the array. pBinSize will point to the value in the size map for this polygon's subset so that we can increment the size as we add indices to the index array. This means, with each polygon we add, pBinSize will always tell us how many indices we have currently collected in this subset's index array and therefore, at what location to place the new indices.

Our next task is to fetch the leaf bin that the polygon will be assigned to. We pass the polygon's attribute ID into the GetLeafBin method to retrieve a pointer to the leaf bin that is assigned to the polygon's subset and will ultimately receive its indices.

```
            // Retrieve the render data item for this leaf / attrib ID
            pLeafBin    = GetLeafBin( pPoly->m_nAttribID );
            pRenderData = pLeaf->GetRenderData( pPoly->m_nAttribID );
```

Notice in the above code how we also call the leaf's GetRenderData method to fetch the RenderData structure in the leaf for the current subset. Remember, there will ultimately be a RenderData structure stored in the leaf for every subset contained in that leaf as it is inside the RenderData structure that the leaf's triangle range (for that subset) is stored. Of course, at this point, the leaf may not have a RenderData structure allocated for it if this is the first polygon we have encountered in the current leaf that has this subset. Therefore, if a RenderData structure exists in the leaf for the passed subset, its pointer will be stored in pRenderData. If not, pRenderData will be assigned a value of NULL, which means we had better allocate one.

```
            // If we were unable to find a render data item yet built for this
            // leaf / attribute combination, add one.
            if ( !pRenderData )
            {
                pRenderData = pLeaf->AddRenderData( pPoly->m_nAttribID );
                if ( !pRenderData ) return false;

                // Store the leaf bin for later use
                pRenderData->pLeafBin = pLeafBin;

            } // End if no render data
```

As you can see in the above code, if the current leaf being processed does not yet have a RenderData structure allocated for it for the subset of the current polygon being processed, we allocate a new one in that leaf by using the CBaseLeaf::AddRenderData method and passing in the subset ID/leaf bin we would like it associated with. The function will return a pointer to a new RenderData structure which we

will store in the pRenderData local pointer. We will then use this pointer to assign the RenderData's pLeafBin member to point at the leaf bin that will contain its indices (which was fetched above)

At this point we have the polygon we are currently processing and we also have the RenderData structure associated with its subset for the current leaf we are processing. The m_nVBCount local variable describes the index of the vertex buffer we are currently collecting vertices and indices for. This will obviously be 0 at the start of the function since we are collecting vertices for vertex buffer[0] in the tree's vertex buffer array. Every time the vertex array is filled up and the CommitBuffers method is called to create the vertex and index buffers, this value will be incremented. For example, after the first set of buffers has been created, it will contain an index of 1 and we will know that any vertices we collect are going to be assigned to the tree's second vertex buffer. We also know that the next set of index buffers we create for the leaf bins will index into this second buffer.

Each RenderData structure in a leaf (remember, there will be one for each subset stored in that leaf), will contain an array of one or more Element structures. For example, if a leaf has a series of polygons belonging to the same set that ultimately ends up getting spread over two vertex buffers, the RenderData structure for that subset (in that leaf) will have two Element structures. Each Element contains the run of triangles for one of those vertex buffers.

So now that we know which RenderData item this polygon's triangles need to be added to, we have to get the Element structure within it for the current vertex buffer being compiled. In the next section of code we loop through each Element in the RenderData's element array to search for the one that has a vertex buffer index assigned to it that is equal to the vertex buffer we are currently compiling data for (m_nVBCount). When we find the correct element to add the polygon's triangles to, we break from the loop as shown below.

```
            // Search for element associated with the current vertex buffer
            for ( j = 0; j < pRenderData->ElementCount; ++j )
            {
                pElement = &pRenderData->pElements[j];
                if ( pElement->VBIndex == m_nVBCount ) break;

            } // Next Element
```

If, at the end of the above loop, the loop variable *j* is equal to the RenderData's current element count, it means the loop did not break early. This tells us that the RenderData structure does not currently have an Element for the vertex array currently being compiled. This also means the current polygon is the first we have found in this leaf that uses this subset since collecting data for the current vertex buffer. Thus we will have to add a new Element to the RenderData's element array.

The following code adds a new Element structure to the leaf's RenderData structure that matches the attribute ID of the polygon currently being processed. This is all handled in the CBaseLeaf::AddRenderDataElement function. It finds the RenderData structure in its array with an attribute that matches the polygon's and adds a new Element structure to the end of its element array. It then returns a pointer to this new element.

```
            // If we reached the end, then we found no element
```

```
            if ( j == pRenderData->ElementCount )
            {
                // Add a render data element for this vertex buffer
                pElement = pLeaf->AddRenderDataElement( pPoly->m_nAttribID );
                pElement->IndexStart     = *pBinSize;
                pElement->PrimitiveCount = 0;
                pElement->VBIndex        = m_nVBCount;

            } // End if no element for this VB
```

Notice in the above code how once we get back the new Element for the current vertex buffer, we set its primitive count to 0 as no triangles have been added to it yet. We also set its VBIndex member to the current value stored in m_nVBCount. This tells us the number of vertex buffers we have created so far and therefore, the index of the current vertex buffer we are collecting data for.

What requires a little explanation is why we set the IndexStart value of this element to the value currently contained in pBinSize. We saw just inside the polygon loop that this is used to point at the value in the row of the BinSizes map that records the current number of indices that have been collected so far for this attribute. This was set to zero for every subset at the beginning of the leaf loop and will be incremented every time we add indices to the corresponding index array. Therefore, the value pointed at by this pointer will always contain the number of indices we have added to that subset's temporary index array so far. Therefore, we know that the indices we are about to add for the current polygon we are processing will begin at that location in the buffer. To clarify, if the pBinSize pointer points to a value of 6 and the polygon we are currently processing belongs to subset 20, it means we have current added 6 indices to the temporary index array stored at BinIndexData[20]. When we add the current polygon's indices to this same array, and because we only enter the above section of code if we have just created a new element, this triangle must be the first triangle we have found in the leaf so far for the current vertex buffer being compiled. Thus, its first index will be the start index for the run of triangles that use this subset and may exist in this leaf.

At this point we have the polygon itself, the leaf bin in which it is contained, the RenderData structure within that leaf that pertains to the subset of the polygon, and the Element structure in that RenderData structure that references the current vertex buffer being compiled. We are now ready to start adding the vertices and indices in this polygon to their respective buffers.

Before we can add the vertices of any of the polygon's triangles to the vertex array (pVertices), we first make sure that there is room enough to do so. If the current number of vertices we have collected in the vertex array so far (nVertexCount) is greater than the maximum number of vertices we will be able to store in a single vertex buffer (nMaxVertexCount) minus 3 (we need at least three spare slots to add the next triangle), it is time to commit the vertex array and all the index arrays we have compiled for each leaf bin to Direct3D vertex and index buffers.

However, before we do, we will perform a weld on the vertex array just to make sure that we cannot compact the vertex data and make some more room in there. Therefore, in the next section of code, once we realize there is no more room left in the vertex array for another triangle, we call the CBaseTree::WeldBuffers method. As the job of this method is to remove duplicated vertices (and remap the indices accordingly), the function returns the new vertex count describing how many vertices are in

the passed vertex array after the weld has been performed. Since the function will also have the job of remapping all the index arrays that reference vertices that get collapsed in the weld process, we must also pass this function the map of index arrays (for each leaf bin).

```
            if ( nMaxVertexCount != nTotalVertices &&
                 nVertexCount > nMaxVertexCount - 3 )
            {
                // Weld the data built so far
                nVertexCount = WeldBuffers( nVertexCount,
                                            pVertices,
                                            BinIndexData,
                                            BinSizes );
```

We will look at the code to this function later, so for now just know that when it returns, the pVertices array may have had vertices removed. The return value (which we store in nVertexCount) describes the new number of vertices in this array. On function return, any index arrays stored in the BinIndexData map will have had some of their indices remapped to new vertex positions in the array if these indices referenced vertices that were collapsed during the weld operation.

After the weld has been performed it is entirely possible that we may have made more space in the vertex buffer and may now have enough space to add the triangles in this polygon. However, if the vertex count of the pVertices array is still too large and we cannot add another triangle, we must commit the buffers using the CBaseTree::CommitBuffers method as shown below.

```
                // If there is still not enough room commit buffers
                if ( nVertexCount > nMaxVertexCount - 3 )
                {
                    CommitBuffers( nVertexCount,
                                   pVertices,
                                   BinIndexData,
                                   BinSizes,
                                   b32BitIndices );


                   nVertexCount = 0;

                    // Add a new render data element and prepare it
                    pElement = pLeaf->AddRenderDataElement( pPoly->m_nAttribID );
                    pElement->IndexStart     = 0;
                    pElement->PrimitiveCount = 0;
                    pElement->VBIndex        = m_nVBCount;

                } // End if start a clean buffer.

            } // End if not enough room for even a single triangle.
```

The CommitBuffers method is passed the vertex array that we have compiled so far and the map containing all the index arrays we have compiled for this vertex array so far. It is also passed the size of the vertex array and the map describing the size of the index arrays contained in the BinIndexData map. Note that we also pass the Boolean that describes to this function whether or not it should create 32-bit or 16-bit index buffers.

The CommitBuffers method is not a large method, but it has a very important job. For starters, it creates a new vertex buffer and populates it with the data in the passed vertex array and then adds this vertex buffer to the tree's vertex buffer array. It will also loop through every row in the BinIndexData map and extract the index arrays compiled for each leaf bin (for the current vertex buffer). For each index array, it will create a new index buffer and populate it. Then it will add each index buffer to the correct leaf bin which should own it. After the index buffer for each leaf bin has been created, its corresponding size value in the BinSizes map will be reset to zero so that we can start collecting new indices for the next vertex buffer. This function will also increase the tree's m_nVBCount variable so that it reflects the new number of vertex buffers stored in the tree.

When the function returns, we wish to start the collection process all over again (the index array counters have already been reset back to zero in the BinSizes map by the CommitBuffers function). Therefore, we set the current vertex count back to zero so that the vertex array can be used again to collect vertices for the next vertex buffer. This means that any Elements we create from this point on (in the RenderData structures of each leaf) will describe triangle runs in the new vertex buffer we are about to build.

Finally, notice in the above code that once we have committed the buffers, we have essentially deduced that the polygon being processed will be added to a new Element in the leaf's RenderData structure (describing triangle runs in the new vertex buffer we are just about to compile). Therefore, we add a new Element to the RenderData structure that is linked to the new vertex buffer we are about to create. Remember, the CommitBuffers method would have incremented the value of m_nVBCount so that it now contains the index of the next vertex buffer we will create the next time CommitBuffers is called. We still have not added the polygon's indices to this Element yet, so we set the primitive count of this new structure to 0. Further, because we have just committed the buffers, we know this polygon's triangles will be positioned at the start of the index buffer that will be created for the new vertex buffer (for the leaf bin associated with the polygon attribute). Therefore, we can set the index start to zero as well.

As discussed many times before, we will add the polygon triangles to the index buffer by stepping around each vertex in the polygon. For each vertex we will add it to the vertex array and for every vertex beyond the second one, we will create indices for the first vertex in the polygon, the previous vertex in the edge, and the current vertex. These three indices will be added to the index array associated with this leaf bin and the pBinSize pointer will have the value it points at (the number of indices in the array) incremented each time an index is added.

In the first section you will see us setting up a loop to step through each vertex in the edge. Notice just above the loop that we store the number of vertices currently in the vertex array in the FirstVertex local variable. This is important because we will need to keep track of where this polygon's vertices are being added from so that we can construct the right indices that index them.

```
            // Store triangle pre-requisites
            ULONG FirstVertex    = nVertexCount;
            ULONG PreviousVertex = 0;

            // For each triangle in the set
            for ( j = 0; j < pPoly->m_nVertexCount; ++j )
```

```
            {
                // Add this vertex to the buffer
                pVertices[ nVertexCount++ ] = pPoly->m_pVertex[ j ];
```

If the current loop variable *j* is greater then or equal to 2, it means we have added at least three vertices from this polygon so far and we can start generating indices to describe its triangles. As Figure 15.4 illustrates, at vertex 3 we will add the indices for vertices 1, 2, and 3. At vertex 4 we will increment the previous vertex and the current vertex so that the next triangle indexes vertices 1, 3, and 4, and so on around the edges of the polygon.

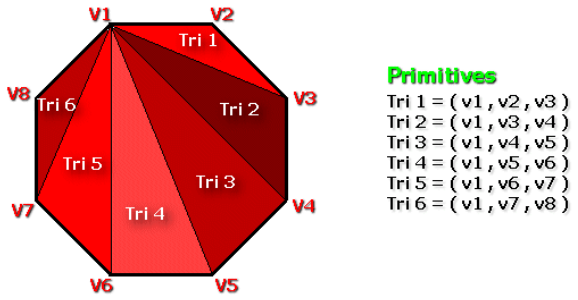


Figure 15.4

For every three indices we add to the index buffer we also know that we have just added a triangle (primitive) to the current element, so we increment the primitive count for that element. We then adjust the previous vertex by 1 so that it now points at the current vertex so that in the next iteration of the loop, we will have the next pair of vertices in the edge.

```
                // Enough vertices added to start building triangle data?
                if ( j >= 2 )
                {
                    // Add the index data
                    pIndexData[ (*pBinSize)++ ] = FirstVertex;
                    pIndexData[ (*pBinSize)++ ] = PreviousVertex;
                    pIndexData[ (*pBinSize)++ ] = nVertexCount - 1;

                    // Update leaf element, we've added a primitive
                    pElement->PrimitiveCount++;

                } // End if add triangle data

                // Update previous vertex
                PreviousVertex = nVertexCount - 1;
```

Because we are adding the polygons to our index buffers one triangle at a time, we may find that we might be only part of the way through adding its triangles when we fill up the vertex buffer. Looking at Figure 15.4 for example, we can see how the first vertex might be v1, the previous vertex might be v4 and the current vertex might be v5, which collectively describe triangle 3. However, after adding triangle 3, we might find that the vertex buffer is full and we have to switch buffers. If this is the case, we will first call the weld function to try compacting the vertex array, hopefully making enough room to add all the vertices in this polygon to the vertex array. Notice this time that we pass in two additional parameters to the WeldBuffers function.

```
                if ( nMaxVertexCount != nTotalVertices &&
                     nVertexCount == nMaxVertexCount )
                {
                    // Weld the data built so far
                    nVertexCount = WeldBuffers( nVertexCount,
                                                pVertices,
```

```
                                                BinIndexData,
                                                BinSizes,
                                                &FirstVertex,
                                                &PreviousVertex );
```

Because we are in the middle of stepping around the vertices of a triangle when we perform the weld, we will need to know where the position of the first vertex and the previous vertex in the edge got moved to during the weld. This tells us which vertices to use for the next triangle. For example, imagine we have just added triangle 3 in Figure 15.4 where the first vertex (v1) was at position 20 in the vertex buffer and the previous vertex (which was v4 when the triangle was added and was then increased to v5) was set to 24. We know that if the weld had not been performed, the next triangle added would consist of vertices 20, 24, and 25. However, after the weld, vertices 20 and 24 might get repositioned in the buffer to locations 10 and 14, in which case we will need to update the values stored in FirstVertex and PreviousVertex to these values so that the next triangle added will use vertices 10 and 14 and the next vertex in the edge. That is what the last two values to the weld function are for. The weld function will return the new positions of these vertices using these output parameters. That is, FirstVertex and PreviousVertex tell the weld function the two vertices we wish to track the changes for. After the weld is complete, these values will be updated with the new vertex positions.

Of course, if the weld function did not successfully make enough room in the vertex array for the next vertex in the polygon, we will need to commit the buffers and start collecting vertices and indices for the next vertex buffer.

```
                    // If there is still not enough room, commit the changes
                    if ( nVertexCount == nMaxVertexCount )
                    {
                        // Commit the changes
                        CommitBuffers( nVertexCount,
                                       pVertices,
                                       BinIndexData,
                                       BinSizes,
                                       b32BitIndices );

                        nVertexCount = 0;
```

What we have to remember is that if a polygon spans multiple buffers, we will need to also add the first vertex and previous vertex to the new buffer as well. For example, imagine in Figure 15.4 we have added triangle 1 and then we start a new buffer. The next vertex in the polygon to be processed would be v4 and the triangle that should be added to the new buffer would be <v1, v3, v4>. However, v1 and v3 exist in the previous vertex buffer so they cannot possibly be indexed by the new index buffers we are about to create. Therefore, we must make sure that these vertices exist in the new buffer as well.

In the next section of code we can see that if we do change buffers mid-polygon, we add a new element for that vertex buffer in the RenderData structure associated with the polygon's subset. We also assign its VBIndex member so that it describes the next vertex buffer that will be created the next time CommitBuffers is called. As we have not added any primitives yet, we will set its index start and primitive count to zero.

```
                        // If the polygon spans multiple VB's,
                        // then we need a new element
                        if ( (signed)j < pPoly->m_nVertexCount - 1 )
                        {
                            // Add a new render data element and prepare it
                          pElement=pLeaf->AddRenderDataElement(pPoly->m_nAttribID);
                          pElement->IndexStart     = 0;
                          pElement->PrimitiveCount = 0;
                          pElement->VBIndex        = m_nVBCount;
```

We will now add the first vertex in the polygon and the current vertex (which will become the previous vertex in the next iteration) to the first element of the new vertex array. We will then set this vertex array's count to 2 since we have only collected two vertices for this vertex array so far. We then assign the FirstVertex and PreviousVertex variables to contain the indices of the first and second vertices in the new vertex array so that in the next loop, the vertices we just duplicated in the new buffer are used to construct the next triangle. The remainder of the entire process is shown below.

```
                            // Since this polygon data is now split over multiple
                            // vertex buffers, we're going to need to re-add the
                            // first and previous vertices to the
                            // new buffer as well. This is so that we can form a
                            // valid triangle next time assuming there is any
                            // triangle data left to be processed for this polygon,

                            pVertices[0]   = pPoly->m_pVertex[ 0 ];
                            pVertices[1]   = pPoly->m_pVertex[ j ];
                            nVertexCount   = 2;
                            FirstVertex    = 0;
                            PreviousVertex = 1;

                        } // End if more triangles to come

                    } // End if not enough room.

                } // End if buffer full

            } // Next Triangle

        } // Next Polygon

    } // Next Leaf
```

At this point in the function we have processed every leaf and every polygon in the tree. We will have constructed vertex buffers and added them to the tree and we will have added (potentially) multiple index buffers to each leaf bin. Every leaf will contain a RenderData structure for each subset it contains, which in turn will contain an array of elements for every vertex buffer that subset (within that leaf) is contained within.

Of course, as the buffers are only committed once they are full, unless the very last vertex we added caused the buffers to be committed, there will still be vertices in the pVertices array and index arrays in the BinIndexData map that have not yet been committed. Therefore, as our final step, we weld any

vertices we have not yet committed before committing them too, creating one last vertex buffer and one last set of index buffers (one per leaf bin which has indices in the index arrays for this vertex buffer).

```
    // Build any remaining data
    if ( nVertexCount > 2 )
    {
        // Weld any remaining data and commit it
        nVertexCount = WeldBuffers( nVertexCount,
                                    pVertices,
                                    BinIndexData,
                                    BinSizes );

        CommitBuffers( nVertexCount,
                       pVertices,
                       BinIndexData,
                       BinSizes,
                       b32BitIndices );

    } // End if any data left to commit
```

With our render data constructed we must release all temporary memory that was not allocated on the stack. So we will iterate through each row in the BinSize map (one for each leaf bin) and extract its attribute ID and index array size. If this size if greater then zero, it means there is a corresponding index array in the BinIndexData map that needs to be deleted from memory. Thus we fetch the index array from the BinIndexData map with the matching attribute ID and call delete on its pointer.

```
    // Release temporary index buffer arrays
    BinSizeIterator = BinSizes.begin();
    for ( ; BinSizeIterator != BinSizes.end(); ++BinSizeIterator )
    {
        ULONG AttribID = BinSizeIterator->first;
        ULONG Size     = BinSizeIterator->second;
        if ( Size > 0 && BinIndexData[ AttribID ] )
            delete []BinIndexData[ AttribID ];

    } // Next Bin
```

We now delete the vertex array that was used to collect the vertices for each vertex buffer before returning a successful build indicator.

```
    // Release temp vertex array.
    if ( pVertices ) delete []pVertices;

    // Success!
    return true;
}
```

This was some very complicated code to try to absorb in one read, so fully expect to have to make a few passes over the source code to this function (and the explanations included here) to fully understand the relationships between the various components.

One of the major processes in the above function that we have yet to cover is the call to the CommitBuffers method. The code to this method will be discussed next.

CommitBuffers - CBaseTree

The function is passed an array of vertices which must be used to create a new vertex buffer for the render system. It will also be passed two STL maps. The first map's rows will contain an index array for each leaf bin. The number of rows in this map will be equal to the number of leaf bins used by the tree. This function will create an index buffer for each of these index arrays and store each index buffer in the relevant leaf bin. The second map that is passed (parameter four) describes the number of indices contained in each of the index arrays. The final parameter is a Boolean which tells the function whether the index buffer it creates for each leaf bin should use 16 or 32-bit indices.

The first thing the function does is grow the tree's vertex buffer array making room for an extra vertex buffer pointer at the end of the array. If there are current vertex buffer pointers in this array, this involves the allocation of a new array large enough to hold all the old buffer pointers plus the space at the end for the new buffer. The buffer pointers are then copied over from the old array into the new one and the old array is released. The tree's pointer to its vertex buffer array is then updated to point at this new array and the vertex buffer count is then increased.

```
bool CBaseTree::CommitBuffers( ULONG VertexCount,
                               CVertex * pVertices,
                               map<ULONG, ULONG*> & BinIndexData,
                               map<ULONG, ULONG> & BinSizes,
                               bool b32BitIndices )
{
    ULONG                      ulUsage = D3DUSAGE_WRITEONLY;
    LPDIRECT3DVERTEXBUFFER9 * ppVBBuffer, pVB;
    CVertex                 * pDestVertices;
    UCHAR                   * pDestIndices;
    ULONG                   * pIndices;
    HRESULT                    hRet;
    ULONG                      i;

    // First allocate space for a new vertex buffer
    ppVBBuffer = new LPDIRECT3DVERTEXBUFFER9[ m_nVBCount + 1 ];
    if ( !ppVBBuffer ) return false;

    // Any old data?
    if ( m_nVBCount > 0 )
    {
        // Copy over the old data, and release the old array
        memcpy( ppVBBuffer,
                m_ppVertexBuffer,
                m_nVBCount * sizeof(LPDIRECT3DVERTEXBUFFER9) );

        delete []m_ppVertexBuffer;

    } // End if any old data
```

```
    // Set the new entry to NULL
    ppVBBuffer[ m_nVBCount ] = NULL;

    // Store the new buffer pointer
    m_ppVertexBuffer = ppVBBuffer;
    m_nVBCount++;
```

Notice in the above code that we assign the ulUsage variable the value of D3DUSAGE_WRITEONLY. This variable will be used for the vertex buffer creation flags. We will inform Direct3D to create the vertex buffer in write only mode so that memory placement of the vertex buffer is chosen based on the best performance. If the current device is a software device (which will be stored in the tree's m_bHardwareTnL member passed into its constructor), the D3DUSAGE_SOFTWAREPROCESSING flag will need to be combined with ulUsage so that the vertex buffer is created in system memory (for efficient CPU transformation and lighting).

Next we allocate a new vertex buffer large enough to hold the number of vertices contained in the passed array. Allocation will be in the managed resource pool. We then store the newly create vertex buffer's pointer in the tree's vertex buffer array, as shown below.

```
    // Should we use software vertex processing ?
    if ( !m_bHardwareTnL ) ulUsage |= D3DUSAGE_SOFTWAREPROCESSING;

    // Create the actual vertex buffer ready to store the data
    hRet = m_pD3DDevice->CreateVertexBuffer( VertexCount * sizeof(CVertex),
                                             ulUsage,
                                             VERTEX_FVF,
                                             D3DPOOL_MANAGED,
                                             &pVB,
                                             NULL );
    if ( FAILED(hRet) ) return false;

    // Store the vertex buffer in the array
    // (so that it's released if anything goes wrong)
    m_ppVertexBuffer[ m_nVBCount - 1 ] = pVB;
```

With the new vertex buffer allocated and added to the tree's vertex buffer array, we then lock the vertex buffer and copy in all the vertices in the input vertex array before unlocking it.

```
    // Lock the vertex buffer ready to copy
    hRet = pVB->Lock( 0, 0, (void**)&pDestVertices, 0 );
    if ( FAILED(hRet) ) return false;

    // Copy over the data that we were passed
    memcpy( pDestVertices, pVertices, VertexCount * sizeof(CVertex) );

    // Unlock the buffer, we're done with VB creation
    pVB->Unlock();
```

The vertex buffer has now been dealt with. Next we have to create the index buffers for each leaf bin. First we set up a loop to iterate though each row in the BinIndexData map (one row for each leaf bin)

and extract the attribute ID for that row. This tells us what leaf bin the index array should be assigned to. We also extract the size of the index array (stored in the BinSizes map with the same key) and fetch a pointer to the leaf bin assigned to that attribute using the CBaseTree::GetLeafBin function.

```
    // Now that we've built the VB, we need to propogate the index data
    map<ULONG, ULONG*>::iterator DataIterator = BinIndexData.begin();
    for ( ; DataIterator != BinIndexData.end(); ++DataIterator )
    {
        ULONG AttribID = DataIterator->first;
        ULONG Size     = BinSizes[ AttribID ];

        // Retrieve the leaf bin
        CLeafBin * pLeafBin = GetLeafBin( AttribID );
        if ( !pLeafBin ) { BinSizes[AttribID] = 0; continue; }
```

Every index buffer added to a leaf bin is contained inside its own CLeafBinData structure inside the leaf bin's internal array of these structures. Thus, we allocate a new CLeafBinData structure.

```
        // Allocate a new LeafBinData structure
        CLeafBinData * pData = new CLeafBinData;
        if ( !pData ) return false;
```

We now instruct the leaf bin to add this CLeafBinData pointer to its internal array of CLeafBinData pointers.

```
        // Add to leaf bin, so that the data is released if something goes wrong
        if ( !pLeafBin->AddLeafBinData( pData ) ) { delete pData; return false; }
```

It is time to fill out the information for the CLeafBinData structure, including whether or not it uses 32-bit indices, the number of triangles that will be in the index buffer assigned to this CLeafBinData structure, and the index of the vertex buffer in the tree's vertex buffer array that this index buffer will index into. We also store the number of vertices in that vertex buffer in the structure.

```
        // Fill out the data items for this leaf bin.
        pData->m_b32BitIndices = b32BitIndices;
        pData->m_nFaceCount    = Size / 3;
        pData->m_nVBIndex      = m_nVBCount - 1;
        pData->m_nVertexCount  = VertexCount;
```

If the index array inside the BinIndexData map for this leaf bin has no indices, we will just skip generating an index buffer for this leaf bin (for this vertex buffer) and continue on to process the next leaf bin.

```
        // If there is no data here,
        // we'll just continue (but we must have allocated the leaf bin data)
        if ( Size == 0 ) continue;
```

If we get this far, it indicates that there are indices for the current leaf bin, so we will need to create an index buffer.

```
        // Create the index buffer ready to store the data
        UCHAR     IndexStride = ( b32BitIndices ) ? 4 : 2;
        D3DFORMAT Format      = (b32BitIndices) ? D3DFMT_INDEX32 : D3DFMT_INDEX16;

        hRet = m_pD3DDevice->CreateIndexBuffer( Size * IndexStride,
                                                ulUsage,
                                                Format,
                                                D3DPOOL_MANAGED,
                                                &pData->m_pIndexBuffer,
                                                NULL );
        if ( FAILED(hRet) ) return false;
```

With the index buffer for this leaf bin/vertex buffer combination allocated, we will now lock it and copy over all the index data from the index array for this leaf bin contained in the BinIndexData map.

```
        // Lock the index buffer ready to copy
        hRet = pData->m_pIndexBuffer->Lock( 0, 0, (void**)&pDestIndices, 0 );
        if ( FAILED(hRet) ) return false;

        // Copy over the buffer data
        if ( b32BitIndices )
        {
            // We can do a straight copy if it's a 32 bit index buffer
            pIndices = DataIterator->second;
            memcpy( pDestIndices, pIndices, Size * IndexStride );

        } // End if 32bit
        else
        {
            pIndices = DataIterator->second;
            for ( i = 0; i < Size; ++i )
            {
                // Cast everything down to 16 bit
                ((USHORT*)pDestIndices)[i] = (USHORT)pIndices[i];

            } // Next Index

        } // End if 16bit

        // Unlock the buffer, we're done with IB creation

        pData->m_pIndexBuffer->Unlock();
```

Notice in the above code that when we create the index buffer, we store the returned pointer in the leaf bin's CLeafBinData structure.

Next we assign the address of the vertex buffer we have just created to the CLeafBinData's vertex buffer pointer so that this structure now contains pointers to both the vertex buffer and the index buffer that should be bound to the device in order to draw its render batches. Since we are making a copy of the vertex buffer pointer, we also increase its reference count.

```
        // Store a reference to the VB in the data area to make it easy to process
        pData->m_pVertexBuffer = pVB;
```

```
        pVB->AddRef();
```

Finally, having committed the index array for the current leaf bin to an index buffer, we set the size of this array back to zero so that when the function returns we can start collecting indices from the beginning of this array for the next vertex buffer. That ends the loop that is performed for each leaf bin.

```
        // Reset bin size to 0, we've commited the data
        BinSizes[ AttribID ] = 0;

    } // Next set of index data

    // Success!
    return true;
}
```

When the loop ends, every leaf bin that had indices collected for it for the current vertex buffer will have had an index buffer created for it and assigned to it. With the job done, the function then returns a success flag.

WeldBuffers - CBaseTree

We saw this method being called from the BuildRenderData method discussed previously. It is used to perform a weld on the passed vertex array to (hopefully) compact the data and make some more room in the buffer for additional vertices. We call this function whenever the buffer is found to be full so that we are able to squeeze as many triangles into a single vertex buffer as possible. Fewer vertex buffers results in reduced vertex T&L and a smaller number of index and vertex buffer changes during rendering.

The function accepts six parameters. The first two parameters describe the size and contents of the passed vertex array that is going to be welded by this function. The second two parameters are maps that contain the index arrays collected for each leaf bin which indexed into the vertex array and the size of each leaf bin's index array. We must pass the index arrays collected for each leaf bin because they index into the passed vertex array. If the vertex array is going to be compacted and have duplicated vertices collapsed into a single vertex, the index arrays must also be updated so that any indices in any leaf bins' array that reference these deleted vertices are properly updated to reference the new vertices that the original vertices were collapsed onto. The final two parameters are optional. They allow us to pass the address of two vertex indices that will have their values tracked and remapped by the weld and returned to the caller. For example, if we passed in variables that held the values 12 and 24 respectively, this tells the weld function that we would like to know where the 12^{th} and 24^{th} vertex were repositioned in the vertex array after the weld. If these two vertices were relocated to slots 5 and 6 in the vertex buffer after the weld, on function return, the variable passed as the fifth parameter would contain 5 and the variable passed as the sixth parameter would contain the value 6. We saw why we needed to track the remapping of certain vertices during the BuildRenderData function. If a weld was performed partway through adding the triangles of a polygon to the vertex buffer, we needed to track where the two vertices used in the last triangle (that will also be used in the next triangle) will be positioned after the weld. This allows us to reference these vertices as we add the rest of the triangles in the polygon using their new vertex

buffer positions. All other times that a weld is performed in the BuildRenderData method, NULL is passed for the final two parameters since we only need to track the vertex positions when the weld happens mid-way through adding a single polygon.

Performing a weld seems like it would involve a good amount of code as you can imagine the various conditional tests that would be needed and we would certainly want a fast way of determining which vertices are candidates for collapse. While it would not be very difficult to code from scratch, fortunately D3DX includes a welding function that works with D3DX meshes and it is quite fast. All we will have to do is temporarily build a D3DX mesh using the vertex and index data passed in, and perform a weld on the mesh and extract the vertex and index data back out again into their original arrays which can then be returned to the caller. The mesh can then be released since it was only used temporarily so that we would get access to the D3DX weld functionality. Because the vertex array will be welded and all the index arrays (one for each leaf bin) describe triangles in that vertex array, we can think of the vertex array and all the index arrays combined as describing a single mesh. Therefore, what we will do is create a mesh into which we will copy all the vertex data in the vertex array into its vertex buffer and all the indices from all the index arrays (collected for each leaf bin in the BinIndexData map) into its index buffer. This means we will need to know how large and allocation we will for the temporary mesh's index buffer so that we can inform the mesh creation function of how many triangles it will need to store.

The first thing we do in this function is loop through all the rows in the BinSize map and sum up the sizes of each index array (one per leaf bin). We divide the total size by three (# indices per triangle) so that at the end of the loop we know the total number of triangles we will need to store in the mesh. Remember, this mesh will contain all of the triangles contained in all of the index arrays we have collected for each leaf bin/subset that reference the passed vertex buffer.

```
ULONG CBaseTree::WeldBuffers( ULONG VertexCount,
                              CVertex * pVertices,
                              map<ULONG,ULONG*> & BinIndexData,
                              map<ULONG,ULONG> & BinSizes,
                              ULONG * pFirstVertex /* = NULL */,
                              ULONG * pPreviousVertex /* = NULL */ )
{
    ULONG       i;
    HRESULT     hRet;
    LPD3DXMESH  pMesh;
    ULONG       TotalTriangles = 0;
    ULONG       nFirstIndex = 0xFFFFFFFF, nPreviousIndex = 0xFFFFFFFF, Counter = 0;

    // Total the full amount of triangles in the bin data so far
    map<ULONG,ULONG>::iterator SizeIterator = BinSizes.begin();
    for ( ; SizeIterator != BinSizes.end(); ++SizeIterator )
    {
        TotalTriangles += (SizeIterator->second / 3);

    } // Next Index Buffer
```

At this point the TotalTriangles local variable will contain all the triangles described by every index array. We now create a D3DXMesh that is large enough to store the correct number of vertices and triangles.

```
    hRet = D3DXCreateMeshFVF( TotalTriangles,
                              VertexCount,
                              D3DXMESH_SYSTEMMEM |
                              D3DXMESH_SOFTWAREPROCESSING | D3DXMESH_32BIT,
                              VERTEX_FVF,
                              m_pD3DDevice,
                              &pMesh );

    if ( FAILED( hRet ) ) return false;
```

Notice that since we are using this mesh for CPU bound operations (we do not intend to render it) we pass in the D3DXMESH_SYSTEMMEM and D3DXMESH_SOFTWAREPROCESSING flags so that we have a mesh created in system memory.

Next, we lock its vertex buffer, its index buffer, and its attribute buffer because we will need to fill the mesh with the data from our vertex and index arrays.

```
    ULONG   *pDestIndices, *pAttributes;
    CVertex *pDestVertices;

    // Lock the vertex, index and attribute buffers ready for population
    if ( FAILED( pMesh->LockIndexBuffer( 0, (void**)&pDestIndices ) ))
         { pMesh->Release(); return VertexCount; }

    if ( FAILED( pMesh->LockVertexBuffer( 0, (void**)&pDestVertices ) ))
         { pMesh->Release(); return VertexCount; };

    if ( FAILED( pMesh->LockAttributeBuffer( 0, &pAttributes ) ))
         { pMesh->Release(); return VertexCount; }
```

We copy the vertices in the passed array into the mesh's vertex buffer and then set every entry in the mesh's attribute buffer to zero.

```
    // Copy over the vertices
    memcpy( pDestVertices, pVertices, VertexCount * sizeof(CVertex) );

    // Zero out the attribute buffer
    memset( pAttributes, 0, TotalTriangles * sizeof(ULONG) );
```

Why do we fill the attribute buffer with zeros? We do this because the D3DX weld function may perform operations on the indices that cause them to be moved around in the index buffer. For example, triangles may be shuffled around so that they are grouped by subset. The problem is, although we wish the vertex data to be compacted and the indices to have their values updated to correctly use these vertices, we must be absolutely sure that the weld function never changes the location of the triangles within the mesh's index buffer. We need to know that the triangles stay exactly where they are so that we can extract them out again into the index arrays for each leaf bin. Imagine for example that we added

10 triangles from leaf bin 1 followed by 10 triangles from leaf bin 2 to the mesh's index buffer. Provided the triangles never have their positions within the index buffer changed, we know that after the weld we can extract the first 10 triangles back out into leaf bin 1's array and the second 10 back out into leaf bin 2's index array. However, if these triangles were repositioned, we would lose all knowledge of which triangles in the welded index buffer need to be copied back out into their original index array. By placing all zeros in the mesh's attribute buffer, we trick the mesh into thinking that all the triangles belong to a single subset (subset zero) and therefore, the optimization that moves triangles for better subset batching will not be performed.

Our next task is to loop through every row in the BinIndexData map and extract the index pointer for each subset/leaf bin. We then copy the indices stored in each index array into the mesh's index buffer. The next section of code starts the beginning of this loop.

```
    // Now we need to update the indices
    map<ULONG,ULONG*>::iterator BinIterator = BinIndexData.begin();
    for ( ; BinIterator != BinIndexData.end(); ++BinIterator )
    {
        ULONG AttribID = BinIterator->first;
        ULONG Size     = BinSizes[ AttribID ];

        // Skip if there is no data
        if ( Size == 0 ) continue;

        // Copy over the indices
        ULONG * pIndices = BinIterator->second;
        memcpy( pDestIndices, pIndices, Size * sizeof(ULONG) );
```

At this point in the current iteration of the loop we have copied over all the indices of the current index array we are processing into the mesh's index buffer (pDestIndices).

The next section of the loop code may look a little strange, so let us quickly explain why it is needed. As the final two input parameters, the caller can pass the address of two unsigned longs that describe vertex indices into the original vertex array passed. We know that the weld function will automatically update all the indices in the index buffer so that the triangles correctly index the updated vertices in the post-weld vertex buffer (even if many of the original vertices were deleted/collapsed). However, if we call the weld function partway though adding the triangles of a single polygon to the vertex array (which we might do in the BuildRenderData method), we will need to track the position changes made to two of the polygons vertices. This way, when the weld function returns, we can continue to add another triangle using the two vertices used in the previous triangle that was added before the weld function was called. We explained where and why we need this functionality when we covered the BuildRenderData method earlier, so refer back to that discussion if you are fuzzy on the details.

If either pFirstVertex or pPreviousVertex are not NULL, it means they point to variables assigned by the caller that contain the indices of two vertices. It also means that the caller would like to know where these two vertices end up in the vertex array after the weld. Unfortunately, the weld function does not give us this information, so we will have to do a little work beforehand. Essentially, as we add each index array to the index buffer, we will loop through those indices searching for an index that references the vertex we are interested in tracking the position changes for. For example, if *pFirstVertex=10 it

means the caller would like to know the new index for vertex 10 if it got moved in the vertex array during the weld. In order to do this, we test each index array as we add it and search for the first index we find that references this vertex. We then record the location of this index. After the weld, the vertices may have changed, but the index we recorded will still be in the same location in the index buffer, so we can read back the value it now contains. This will tell us the vertex that index has been updated to use and the new position of the original vertex 10 in the new vertex buffer.

Let us just step through an example so that the logic is clear. Let us say that the caller would like to know the post-weld positions of vertex 10 and vertex 12. They would pass in as the pFirstVertex and pPreviousVertex parameters pointers to two variables that contain the values 10 and 12, respectively. In this next section, we will search for the first index in any index buffer that currently indexes vertices 10 and 12 in the vertex array and we will record the locations of these indices in the index buffer. Let us imagine that we found that the first index that references vertex 10 is in leaf bin 2's index array at location 100 in the mesh's main index buffer. We will record this index location. Let us also assume that we find the first reference to vertex 12 in leaf bin 5 and record the location where it has been added to the mesh's index buffer as well. Assume that it has been placed at index 500 in the mesh's index buffer. So we have the following.

*pFirstVertex = 10
*pPreviousVertex = 12

nFirstIndex = 100
nPreviousIndex = 500

Just to be clear, we have determined that there is an index that references vertex 10 stored at location 100 in the mesh's index array and an index that references vertex 12 at location 500 in the mesh's index array. Once we have recorded these values, we no longer need to search for it anymore; we are just interested in finding the first occurrence of these two vertices in the index buffer. After the weld is performed, vertices 10 and 12 may be in completely different locations. However, the triangles (and thus their indices) in the mesh index buffer will not have changed position at all, but they will have had their values updated by the weld function so that they correctly reference the updated vertex buffer. Therefore, all we have to do is read back the values now stored in the index buffer at locations 100 and 500 and we will have the new locations of vertex 10 and 12 in the vertex buffer. We can then return this information to the caller.

Let us look at the code in this loop that searches and records the index locations prior to the weld being performed. Notice that we only perform the search if we have not already found the first index that references each vertex.

```
        if ( (pFirstVertex && nFirstIndex == 0xFFFFFFFF) ||
             (pPreviousVertex && nPreviousIndex == 0xFFFFFFFF) )
        {
            // Loop through the indices
            for ( i = 0; i < Size; ++i )
            {
                if ( (pFirstVertex &&
                      nFirstIndex == 0xFFFFFFFF)
                      && pIndices[i] == *pFirstVertex ) nFirstIndex = Counter + i;
```

```
                if ( (pPreviousVertex &&
                      nPreviousIndex == 0xFFFFFFFF)
                      && pIndices[i] == *pPreviousVertex) nPreviousIndex=Counter+i;

            } // Next Index

        } // End if search for remap
```

At the start of the code, nFirstIndex and nPreviousVertex are set to 0xFFFFFFFF which means we have not yet found the location of an index that references the vertex. Notice we only step into this code block if we have not yet found both nFirstVertex and nSecondVertex and if the application did not pass NULL as the pFirstVertex and pPreviousVertex parameters. Inside the code block we loop through every index in the index array we are about to add and test to see if its value is equal to pFirstVertex. If it is, then we have found an index that references the first vertex we are interested in tracking, so we record the location of that index in the nFirstIndex variable. As this variable will no longer be set to 0xFFFFFFFF, the search for this index will not be performed again when adding any other vertex arrays. We do exactly the same thing for the pPreviousVertex and record the location of the first index we find that references it in nPreviousVertex. Notice above that the Counter variable starts at zero and is incremented at the bottom of the loop by the size of the index array we just added to the mesh's index buffer. Therefore, at the beginning of each loop iteration, it contains the location of first index in the index buffer of the index array we just added. Thus, Counter + i describes exactly where the index we have just found will be located in the mesh's index buffer (pre- and post-weld).

Finally, at this point in the loop we have added the current index array to the index buffer and have performed a search through those indices for the tracked vertices, so we increment the index buffer pointer past the new indices we just added so that it points at the location where the next index array that we process in the next iteration of the loop should be added. We also increment the Counter variable by the size of the index array so that it always describes (at the beginning of each loop iteration) the location in the index buffer where the index array that has just been added begins.

```
        // Move the destination index array on for the next batch
        pDestIndices += Size;
        Counter      += Size;

    } // Next Index Buffer
```

Once this has been done, we unlock all the buffers in the mesh since it now contains all the vertex and index data needed to perform the weld.

```
    // Unlock the buffers
    pMesh->UnlockIndexBuffer();
    pMesh->UnlockVertexBuffer();
    pMesh->UnlockAttributeBuffer();
```

You will recall from Chapter 8 that the D3DXWeld function must be passed a D3DXWELDEPSILONS structure that allows us to control how fuzzy the comparison is between two vertices. This allows us some control over what are considered duplicate vertices. We can assign the various vertex components

different epsilons so that two vertices that are almost the identical, but have positions that vary by as little as 0.01 (for example) are considered to be the same and are collapsed into a single vertex. Another example would be two vertices that share the exact same location and have normals that are similar enough to warrant welding them into a single vertex.

The D3DXWELDEPSILONS structure is shown below as a reminder. The larger the epsilon we assign to a vertex component, the more fuzzy the compare will be when determining whether two vertices are the same for that component.

```
typedef struct _D3DXWELDEPSILONS
 {
    FLOAT Position;
    FLOAT BlendWeights;
    FLOAT Normal;
    FLOAT PSize;
    FLOAT Specular;
    FLOAT Diffuse;
    FLOAT Texcoord[8];
    FLOAT Tangent;
    FLOAT Binormal;
    FLOAT Tess Factor;
} D3DXWELDEPSILONS;
```

We will set all the members of this structure to a standard epsilon of 0.001. Rather than assign each value of the structure to 0.001 individually, we know that it really is just a block of memory representing 10 floats, so we will just instantiate the structure, access it via a float pointer, and loop through the 10 floats assigning each location a value of 0.001 (1e-3).

```
    // Set all epsilons to 0.001;
    D3DXWELDEPSILONS WeldEpsilons;
    float * pFloats = (float*)&WeldEpsilons;
    for ( i = 0; i < sizeof(D3DXWELDEPSILONS) / sizeof(float); i++ )
         *pFloats++ = 1e-3f;
```

Now we pass this structure, along with the mesh pointer itself, into the D3DXWeldVertices method to perform the weld.

```
    // Weld the vertex data
    hRet = D3DXWeldVertices( pMesh,
                             D3DXWELDEPSILONS_WELDPARTIALMATCHES |
                             D3DXWELDEPSILONS_DONOTSPLIT,
                             &WeldEpsilons, NULL, NULL, NULL, NULL );

    if ( FAILED(hRet) ) { pMesh->Release(); return VertexCount; }
```

The D3DXWELDEPSILONS_WELDPARTIALMATCHES flag instructs the function that we would like duplicated vertices removed. The D3DXWELDEPSILONS_DONOTSPLIT flag instructs the function to never split (i.e., duplicate) vertices that are in separate attribute groups. This is passed just to be safe, since splits should never happen because we have zeroed out the attribute buffer. As far as the mesh is concerned, it only has a single subset.

With the weld performed, we now need to exact the mesh data back out into the vertex and index arrays that were passed into the function so that the modified data is returned to the caller. The first step is to lock the mesh vertex and index buffers and update the VertexCount parameter so that it now reflects the number of vertices in the welded mesh.

```
    // Lock the vertex and index buffers ready for extraction
    if ( FAILED( pMesh->LockIndexBuffer( 0, (void**)&pDestIndices ) ))
         { pMesh->Release(); return VertexCount; }

    if ( FAILED( pMesh->LockVertexBuffer( 0, (void**)&pDestVertices ) ))
         { pMesh->Release(); return VertexCount; };

    // Store updated vertex count
    VertexCount = pMesh->GetNumVertices();
```

In the next section we copy the vertices from the mesh's vertex buffer into the passed pVertices array. This overwrites the old vertex data contained there. We then access the indices in the index buffer at locations nFirstIndex and nPreviousIndex to retrieve the new locations of vertices described by the pFirstVertex and pPreviousVertex parameters. The variables addressed by these two pointers have their values updated to contain the new post-weld position of these vertices, which will be accessible to the caller on function return.

```
    // Copy over the vertices
    memcpy( pVertices, pDestVertices, VertexCount * sizeof(CVertex) );

    // Update remap information if it was requested
    if ( pFirstVertex    ) *pFirstVertex    = pDestIndices[ nFirstIndex ];
    if ( pPreviousVertex ) *pPreviousVertex = pDestIndices[ nPreviousIndex ];
```

Finally, all that is left to do is iterate through each index array in the BinIndexData map and overwrite their indices with the modified mesh indices.

```
    // Now we need to copy back the index buffers
    BinIterator = BinIndexData.begin();
    for ( ; BinIterator != BinIndexData.end(); ++BinIterator )
    {
        ULONG AttribID = BinIterator->first;
        ULONG Size     = BinSizes[ AttribID ];

        // Skip if empty
        if ( Size == 0 ) continue;

        // Copy over the indices
        memcpy( BinIterator->second, pDestIndices, Size * sizeof(ULONG) );

        // Move the destination index array on for the next batch
        pDestIndices += Size;

    } // Next Index Buffer

    // Unlock the buffers and release the mesh
    pMesh->UnlockIndexBuffer();
```

```
    pMesh->UnlockVertexBuffer();
    pMesh->Release();

    // Return new vertex count.
    return VertexCount;
}
```

At the end of this process, we unlock the mesh's buffers and release the mesh before returning the new modified vertex count of the passed vertex array back to the caller.

We have now covered all the code involved in preparing the data for rendering and all of the processes invoked by the CBaseTree::BuildRenderData function. When the BuildRenderData method returns, all the tree's vertex buffers and leaf bins will have been constructed and are ready for use. In the next section we will examine the code that manages the visibility pass on the tree and the code for rendering that visible data.

15.4.3 Processing Tree Visibility / Rendering the Static Data

Before the application instructs the spatial tree to draw any of its subsets, it should first tell it to perform a visibility pass by calling the ISpatialTree::ProcessVisibility method. Because this method is a traversal method that is dependant on node type and the number of children each node has, it must be implemented in the derived classes. That is, the quad-tree's version of this method will step into four children at each node while the oct-tree's version will step into eight.

Although this method must be implemented in the derived class, all that is really in the derived class is the traversal logic that determines whether a given node is within the frustum or not and therefore, whether the children should be traversed. For each leaf that is found to have its bounding box inside the camera's frustum, the ProcessVisibility method will issue a call to the leaf's SetVisible method. As we have seen, it is this method that adds the leaf's triangles to render batches in their respective leaf bins. Thus, we have already done the hard work. All that is left to do in the derived class is the tree traversal and frustum tests at each node.

Because the CBaseTree object must make sure that the render batch lists for each leaf bin are reset prior to the visibility traversal taking place, the derived class's ProcessVisibility method must call the CBaseTree::ProcessVisibility method before starting the visibility pass. Below we show the code to the CQuadTree::ProcessVisibility method.

```
void CQuadTree::ProcessVisibility( CCamera & Camera )
{
    CBaseTree::ProcessVisibility( Camera );

    // Start the traversal.
    UpdateTreeVisibility( m_pRootNode, Camera );
}
```

As you can see it is passed (by the application) the CCamera object whose frustum will be used for the visibility pass. It first calls the base class version of the function prior to calling its own UpdateTreeVisibility method. The UpdateTreeVisibility method is the recursive function that traverses the tree and performs the frustum tests at each node. The ProcessVisibility method in the derived classes is really just a wrapper. In fact, the ProcessVisibility method in all derived classes will look identical to this one so we will not show them all. Each derived class's version of this method will first call the CBaseTree::ProcessVisibility method to reset the batch lists in each leaf bin. Then it will call its version of the UpdateTreeVisibility method to perform the visibility traversal starting at the root node. Throughout this section we will only discuss the implementation in the CQuadTree derived class and you can check the source code for the implementations in the other cases.

ProcessVisibility – CBaseTree

This function is called from the derived class's version of the function to reset the batch lists in each leaf bin. The batch lists are compiled in the CLeafBinData structures in each leaf bin and represent a vertex/index buffer pair. The function loops through each leaf bin and then loops through each CLeafBinData pointer in its internal array. It then resets the CLeafBinData::m_nBatchCount member of each CLeafBinData structure in the leaf bin to zero.

```
void CBaseTree::ProcessVisibility( CCamera & Camera )
{
    LeafBinMap::iterator        BinIterator  = m_LeafBins.begin();
    ULONG                       i;

    // Iterate through the leaf bins and destroy them
    for ( ; BinIterator != m_LeafBins.end(); ++BinIterator )
    {
        CLeafBin * pBin = BinIterator->second;
        if ( !pBin ) continue;

        // For each buffer
        for ( i = 0; i < m_nVBCount; ++i )
        {
            CLeafBinData * pData = pBin->GetBinData( i );
            if ( !pData ) continue;

            // Reset the batch count
            pData->m_nBatchCount = 0;

        } // Next Buffer

    } // Next Leaf Bin

    // Clear the visible leaf list.
    m_VisibleLeaves.clear();
}
```

15.4.4 Traversing the Tree to Ascertain Leaf Visibility

It is clear from the previous discussion that the visibility traversal will be performed in the derived class's UpdateTreeVisibility method. In theory, this method could be extremely simple and just traverse through all the nodes in the tree calling the CCamera::BoundsInFrustum method to test if the current node's bounding volume is within the frustum. If it is not, then it calls the node's SetVisible method to traverse down the rest of that tree and set the visible status all leaves below it to false. If the node is inside the frustum (or partially inside the frustum) we can traverse into its children and test their volumes against the frustum in the same manner. Eventually, we will traverse to all leaves that are inside the frustum, at which point the leaf's SetVisible method will be called. As we have seen, this will cause the triangles in those leaves to be added to the render batches in the leaf bins in which they are contained.

While there is certainly nothing wrong with such a strategy, it does not take full advantage of the hierarchical nature of the tree to reduce the number of plane tests that will need to be performed at each node. Furthermore, it does not take into account the fact that if node was found to be outside a frustum plane in the previous visibility update (outside the frustum), because the movement of the camera will typically be very small between frame updates, the node will probably still be outside that frustum plane in the next update. Therefore, when testing the bounding box of a node against the frustum, we should test the plane it failed on in the last update first in the hopes of rejecting the node immediately without having to perform tests for the rest of the frustum planes. This is referred to as frame coherence. It is essentially the concept of using information collected on a previous frame update to optimize the process in the current one. Let us discuss the frame coherence optimization first.

15.4.5 Frustum Culling with Frame to Frame Coherence

Figure 15.5 depicts a 2D representation of a node's AABB and the current world space position of the camera's frustum planes. As this is a 2D representation, the frustum has four planes instead of the usual six. We can clearly see that the node is not visible because its bounding box is outside the frustum. This obviously means the node's children are also invisible and there is no need to test them against the frustum. Thus, we can immediately mark this node and all its children as invisible.

The planes of the frustum are labeled 1 through 4 and describe the order in which the frustum planes will be tested against the bounding box.

> **Note:** Frustum culling AABBs was discussed in Module I and will not be discussed again here.

If we walk though the frustum test, we will see that it is only when the polygon box is tested against the fourth plane that we know it is outside the frustum.

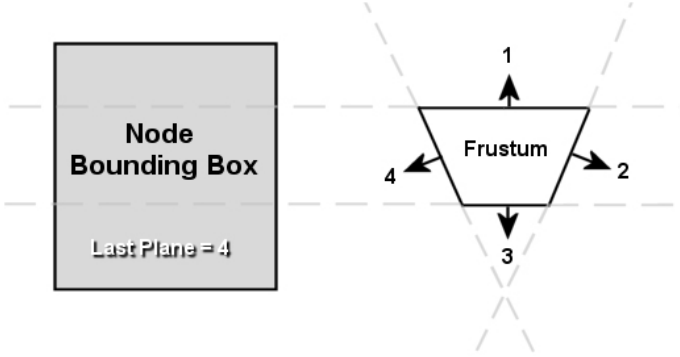


Figure 15.5

For example, we can see that when plane 1 is tested, the box is not totally in front of the plane. Since it spans this plane, the box could very well be in the frustum at this point. Next we test the box against frustum plane 2 where we once again discover the box is totally in its backspace and therefore may still be within the frustum. Remember, we only know that an AABB is inside the frustum if the box is behind all planes (plane normals are assumed to be facing outwards in this example). So far we have tested two planes and have not found a plane the box is totally in front of, so next we move on to plane 3. Once again, the box is not totally in front of this plane, so we know at this point that the box is at least partially contained behind the three planes tested and may well be in the frustum. Finally, we test frustum plane 4 where we discover that the box is totally in its front space and therefore could not possibly be visible. At this point we set the visibility status of all leaves underneath that node to false.

Although this works fine, it took us four plane tests to determine that the box was outside. However, if we knew in advance that we would fail against plane 4, we could have rejected the node with one plane test. Although this might sound like a minimal savings at the node level, just remember that your spatial tree might have thousands of nodes, and the 3D frustum will have six planes. In the worst cases, you could end up having to perform six frustum plane tests at a great many nodes.

If we examine Figure 15.5 again we can imagine that during the next frame update, where the camera will likely only be moved by a very small amount (say, vertically), it would fail on the same plane again. So if we know the plane that we failed on last time, which in this case was frustum plane 4, we could store the index of that plane in the node and make sure that when this node is encountered in the next visibility pass, we test plane 4 first. Although our first visibility pass would require that we test all four planes to learn that the fourth plane was the one that caused the rejection, the next visibility pass (next iteration of the game loop) can be potentially optimized when we visit that node, by reading back the index of the plane we failed on last time and making sure that the CCamera::BoundsInFrustum function test this plane first. In a great many cases, this will lead to immediate rejection of large portions of the tree using a single plane test. Again, if a node was outside a certain frustum plane during one frame update, it will probably be outside that same frustum plane during the next frame update (camera changes are generally minimal between frames when an application is running at 30+ frames per second).

We will implement this frame-to-frame coherency in our visibility system. You now know why each of the node classes we developed in the previous lesson stored that signed char member variable called LastFrustumPlane. As a reminder, we show the CQuadTree node class declaration below.

```
class CQuadTreeNode
{
public:

    // Constructors & Destructors for This Class.
     CQuadTreeNode( );
    ~CQuadTreeNode( );


    // Public Functions for This Class
    void SetVisible( bool bVisible );

    // Public Variables for This Class
```

```
    CQuadTreeNode * Children[4];        // The four child nodes
    CBaseLeaf     * Leaf;               // If this is a leaf, store here.
    D3DXVECTOR3     BoundsMin;          // Minimum bounding box extents
    D3DXVECTOR3     BoundsMax;          // Maximum bounding box extents
    signed char     LastFrustumPlane;   // The frame coherance 'last plane' index.
};
```

When the node is first created, its LastFrustumPlane index will be set to -1 in the constructor. This is because we do not yet have the index of a frustum plane at which this node failed. It will also be set back to -1 for any node that was found to be visible in the previous visibility update. However, for nodes that are currently outside the frustum, or were found to be outside the frustum in the previous visibility update, this member will store a value between 0 and 5, describing the index of the first frustum plane that generated the outside the frustum result. When this node is visited in the next visibility update, and the CCamera::BoundsInFrustum method is called to test the bounding box of the node against the frustum planes, we will pass this plane index into that function to instruct it to test this plane first. Hopefully, the BoundsInFrustum method will immediately determine that the node is still outside this plane and will return the invisible status. If the camera has been moved from the previous position and the node is no longer totally in front of this plane, we will just perform a test on the rest of the planes as normal. If one of the other planes is found to have the AABB totally in its front space, the node's LastFrustumPlane member is updated to store the index of that plane. If the box is not in front of any planes and is therefore inside (or partially inside) the frustum, the node's LastFrustumPlane member is set to -1 and a visible status is returned for the node. The next time this node is encountered in the next visibility process, it will have -1 in its LastFrustumPlane member which means the BoundsInFrustum test will just loop through and test all 6 planes in the usual way.

In a moment we will see how the CCamera::BoundsInFrustum method has been updated to test against the chosen frustum plane first if such a plane index is passed into the function. We will also see how it will update the value of the node's LastFrustumPlane member if a new 'first' rejection plane is found. But before we discuss the code for the frame-to-frame coherence optimization, we will talk about a second optimization that can be introduced to make use of the spatial hierarchy and further reduce the number of frustum plane tests that have to be performed at each node.

15.4.6 Hierarchical Frustum Culling

In the previous chapter we discussed hierarchical frustum culling as the process of stepping through the nodes of the tree and rejecting any node (and all its children) that is outside the frustum from our rendering pipeline without further testing. However, we can further exploit the parent/child relationship of nodes in the tree to introduce another means for frustum plane test reduction in the typical case.

This technique is used to speed up the case where the node is not totally inside or outside the frustum, but is intersecting in such a way that the node's volume is totally behind/inside one or more of the frustum planes. Although we know that in the intersecting case, the children of the node must also be tested against the frustum, we also know (because of the child/parent relationship) that if the parent node is totally behind/inside one of the frustum planes, all of its child nodes/leaves will be as well. Therefore,

when testing the children, we no longer need to test this plane against their volumes because we know that they will always be found to be in the plane's back space.

Figure 15.6 demonstrates this idea in two dimensions by showing a parent node's AABB and the AABBs of its four immediate children.

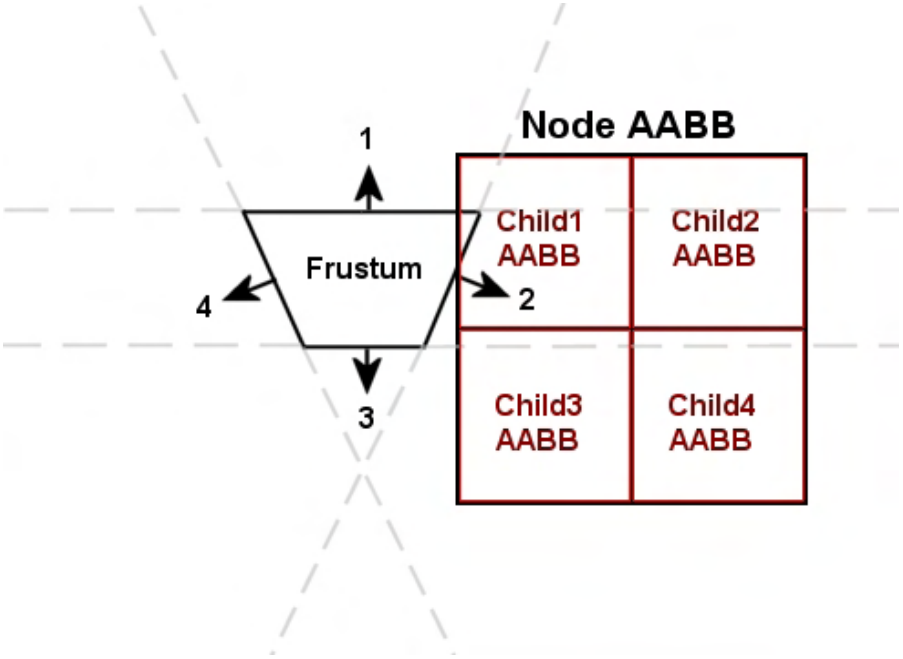


Figure 15.6

The important frustum plane to look at in this example is frustum plane 4. We can see that the parent node's volume intersects frustum planes 1, 2, and 3 and is totally behind frustum plane 4. This tells us that part of the parent node is behind all frustum planes so the node is visible. As the node is not completely inside the frustum, this does not mean that we should set all child nodes to visible automatically. We can see for example, that although the parent node intersects the frustum, three of its children do not. Indeed they are fully outside the frustum. Therefore, if the frustum is intersecting the node, the child nodes must also be tested to determine their visibility status. This means that in a worst case scenario, we have to do six frustum plane tests against the node's volume and perform the same six tests for each of its children, and so on.

If we look at frustum plane 4 in Figure 15.6 we can see that the parent node is completely behind (inside) this plane. But because the parent node is behind plane 4, so are all of its children. This is clear from just looking at the diagram. So we recognize that there is no need to test this plane against the children. The parent can essentially say to the child nodes, "I am behind this plane, so all of you are too. Do not bother performing a test against this plane; just assume that you are behind it and test the other planes instead".

The recursive visibility process will pass an array of bits from parent to child as it traverses the tree. Each bit will represent one of the six frustum planes. The CCamera::BoundsInFrustum function will be updated to take an unsigned char whose first six bits will be used to represent the status of the frustum planes. This unsigned char will be set to zero at the beginning of the traversal.

When a node is reached, its bounding box will be passed into the CCamera::BoundsInFrustum function along with this unsigned char. When this function finds any plane that the AABB of the node is completely behind, it will set the bit to 1 in the unsigned char corresponding to that plane. When the function returns back to the node, the node flag will have the corresponding bits set to 1 for any planes the node's volumes is totally behind. This unsigned char is then passed down to the children. The child

nodes will then pass this unsigned char of frustum plane bits into their CCamera::BoundsInFrustum calls so that that any planes at this node which result in the fully behind case can also have their bits set to 1. The important point here however, is that the CCamera::BoundsInFrustum test will only test planes which have not had their bits set to 1 in the passed unsigned char.

Now that we know the two optimizations we will apply during the frustum culling traversal of the tree, we are ready to look at an example of the UpdateTreeVisibility method. Since these methods are the same in all derived classes (with the exception of having to step into a varying number of children), we will only show the CQuadTree::UpdateTreeVisibility method here.

UpdateTreeVisibility - CQuadTree

This method is called from the CQuadTree::ProcessVisibility function to start the recursive process for the root node. It repeatedly calls itself until all leaves in the tree which intersect the camera's frustum have there visible status set. The function is very simple as nearly all the optimizations are performed inside the modified CCamera::BoundsInFrustum method, which we will discuss in a moment.

The function is passed three parameters. The first is the node currently being visited by the function. This will be the root node of the tree when this function is first called from the CQuadTree::ProcessVisibility method. The second parameter is the camera whose frustum will be used for the visibility test. The third parameter is the function's means of passing the array of 'Totally Inside' frustum bits from one node to the next. Since this parameter is not passed for the root node, the value of the FrustumBits unsigned char will be set to 0 when tree traversal begins at the root.

The first thing the function does is call the CCamera::BoundsInFrustum method, passing in the node's bounding box extents, the unsigned char of frustum plane bits, and the node's LastFrustumPlane member.

```
void CQuadTree::UpdateTreeVisibility( CQuadTreeNode * pNode,
                                      CCamera & Camera,
                                      UCHAR FrustumBits /* = 0x0 */ )
{
    unsigned long i;
    CCamera::FRUSTUM_COLLIDE Result=Camera.BoundsInFrustum( pNode->BoundsMin,
                                                            pNode->BoundsMax,
                                                            NULL,
                                                            &FrustumBits,
                                                          &pNode->LastFrustumPlane);
```

The third parameter to this function is an optional world matrix that can be used to transform the input bounding box into world space prior to performing the test. As our node's bounding box is already in world space, no transformation is required so we set the matrix pointer to NULL.

Before examining the rest of this function, let us just make sure we understand what may have happened when the above function has returned. Any bits set to 1 in the passed FrustumBits char will not be tested

against the frustum. Instead, the function will assume that the node's box is totally behind these planes and will progress to the next plane that needs to be tested. If any of the planes that did require testing (their bits were set to 0) are found to contain the bounding box completely in its back space, that plane will have its bit set to 1 in the FrustumBits char and will also not need to be tested again for any children of this node. Finally, if the BoundsInFrustum method did determine that the node is outside the frustum, the LastFrustumPlane member of the node will have its value altered such that it contains the index of the first plane it failed on. This will be the first plane that is tested by this function the next time the BoundsInFrustum function is called for this node.

The BoundsInFrustum method will return one of three values which are members of the FRUSTUM_COLLIDE enumerated type defined inside the CCamera namespace. This enumerated type is shown below with an explanation of its members.

```
enum FRUSTUM_COLLIDE
{
        FRUSTUM_OUTSIDE     = 0,
        FRUSTUM_INSIDE      = 1,
        FRUSTUM_INTERSECT   = 2,
        FRUSTUM_FORCE_32BIT  = 0x7FFFFFFF
};
```

FRUSTUM_OUTSIDE
The node's bounding volume is totally outside the frustum (the node is not visible). This means we will not perform any more frustum tests down this branch of the tree and we can immediately set the visibility status of any leaves stored under this node to false. If the parent node is completely outside the frustum, so too must be all of its children.

FRUSTUM_INSIDE
The node's bounding volume is contained completely within the frustum (the node is visible). This also means that we do not have to perform any more frustum tests down that branch of the tree and we can immediately set the visibility status of all leaves stored under that node to true. If the parent node is fully contained within the frustum, so too must be all of its children.

FRUSTUM_INTERSECT
The node's bounding volume is partially contained within the frustum (the node is visible). This means we must traverse into the children and continue to perform frustum tests for each child. If the parent node is partially intersecting the frustum, one of more of its children will be visible.

Now that we know what the results mean, let us see the code that processes them in a switch statement.

```
    // Test result of frustum collide
    switch ( Result )
    {
    case CCamera::FRUSTUM_OUTSIDE:
        // Node is not at all visible
        pNode->SetVisible( false );
        return;
```

If the frustum test returned FRUSTUM_OUTSIDE then the node's bounding volume is completely outside the frustum, so the node is not visible and neither are any of its children. When this is the case we simply call the node's SetVisible method passing a visibility parameter of false. This method performs no frustum tests and simply traverses the rest of the branch setting the visible status of any leaves found there to false. The function then returns so that the rest of this branch of the tree avoids further frustum testing.

If the frustum test returned FRUSTUM_INSIDE then the node's bounding volume is entirely contained inside the frustum and therefore, this node and all child nodes are visible and no further frustum tests need to be done down this branch of the tree. Instead we just call the node's SetVisible method passing a visibility status of true. This function will quickly traverse to find all the leaf nodes underneath this current node and will set their visibility status to true. We then return from the function so that we do not perform any more frustum tests down this branch of the tree.

```
    case CCamera::FRUSTUM_INSIDE:
        // Node is totally visible
        pNode->SetVisible( true );
        return;
```

Finally, if the frustum test returned FRUSTUM_INTERSECT it means that the current node is visible but some of its children might not be, so we will need to perform further frustum tests along this branch of the tree. If the current node is a leaf, we set its visible status to true (which we know will cause its triangles to be added to their leaf bin's render batch lists).

```
    case CCamera::FRUSTUM_INTERSECT:
        // We need to resolve this further, unless this is a leaf
        if ( pNode->Leaf )
        {
            pNode->SetVisible( true );
            return;
        } // End if leaf
        break;

    } // End Switch
```

Remembering that the function will have returned already in any case other than FRUSTUM_INTERSECT, this last section of code is also only executed in the intersection case. It simply loops through each child node and recurs into it.

```
    // The remaining case (FRUSTUM_INTERSECT) means we need to test further
    for ( i = 0; i < 4; ++i )
    {
        if ( pNode->Children[i] ) UpdateTreeVisibility( pNode->Children[i],
                                                        Camera,
                                                        FrustumBits );

    } // Next Child
}
```

You will find that the UpdateTreeVisibility method in all the derived classes will be identical to the code shown above, with the exception of the number of children traversed.

SetVisible - CQuadTreeNode

Although we took a brief look at this method in the previous lesson, we will take a quick look at it again now that we have seen it being called in the FRUSTUM_INSIDE and FRUSTUM_OUTSIDE cases in the above function.

```
void CQuadTreeNode::SetVisible( bool bVisible )
{
    unsigned long i;

    // Set leaf property
    if ( Leaf  ) { Leaf->SetVisible( bVisible ); return; }

    // Recurse down if applicable
    for ( i = 0; i < 4; ++i )
    {
        if ( Children[i] ) Children[i]->SetVisible(bVisible);

    } // Next Child
}
```

As you can see, if the node is a leaf it calls its SetVisible method to instruct it to add its polygon data to the render batches in each leaf bin. As we now know, this will also instruct the leaf to add its ‘this’ pointer to the tree’s visible leaf list.

Of course, the final piece of the puzzle is the CCamera::BoundsInFrustum method. Although this method has been in use since Module I in this series, we have now modified it to accept some additional parameters that optimize the frustum rejection for a spatial tree. After we have look at the code to that function, we will have covered all of the code that is executed when the application calls the ISpatialTree::ProcessVisibility method. Thus we will have completely explored how the visibility pass on the tree is performed.

BoundsInFrustum - CCamera

The frustum culling of AABBs and the extraction of the frustum planes in world space was explained in detail in Module I. This function has been with us for quite some time now and we have seen it used in nearly all our lab projects since its inception. Therefore we will not be covering how frustum culling works when discussing this code. If you do not remember how the frustum culling of AABBs can be done, please refer back to Chapter 4 in Module I for the details.

The first thing this function does is call the CCamera::CalcFrustumPlanes method to extract the world space planes for the view frustum and store them in the camera object’s m_Frustum planes array (which

contains 6 elements). We then make a copy of the passed FrustumBits unsigned char into the nBits local variable for ease and speed of access. We also copy the value stored in the passed LastOutside char which will contain the index of the plane which the node that called this function failed on the last time this function was called.

```
CCamera::FRUSTUM_COLLIDE CCamera::BoundsInFrustum( const D3DXVECTOR3 & vecMin,
                                                   const D3DXVECTOR3 & vecMax,
                                                   const D3DXMATRIX * mtxWorld ,
                                                   UCHAR * FrustumBits ,
                                                   signed char * LastOutside )
{
    // First calculate the frustum planes
    CalcFrustumPlanes();

    ULONG           i;
    D3DXVECTOR3     NearPoint, FarPoint, Normal, Min = vecMin, Max = vecMax;
    FRUSTUM_COLLIDE Result  = FRUSTUM_INSIDE;
    UCHAR           nBits = 0;
    signed char     nLastOutside = -1;

    // Make a copy of the bits passed in if provided
    if (FrustumBits) nBits = *FrustumBits;

    // Make a copy of the 'last outside' value to prevent us having to dereference
    if ( LastOutside ) nLastOutside = *LastOutside;
```

If the caller passed a matrix into the function, then it means we were passed a model space bounding box which should be transformed into world space using this matrix before the frustum planes are tested. That is no problem because we wrote a function in the previous lesson that did exactly that.

```
    // Transform bounds if matrix provided
    if ( mtxWorld ) MathUtility::TransformAABB( Min, Max, *mtxWorld );
```

If the node's LastFrustumPlane value (now stored in nLastOutside) is not set to -1, it contains a valid index for a frustum plane and we will test that plane first. However, we will only test it if that plane does not have its bit set to 1 in the nBits array. If it does, then regardless of the fact that the node was found to be outside the plane in the previous visibility pass, it must be inside it now, because one of its parent nodes higher in the tree was found to be completely inside it and set this bit to 1. Alternatively, if a valid last plane index is passed which does not currently have its bit set, we will perform the test.

As the nLastOutside variable contains the index of the plane we want to test first, we will extract the plane normal from the camera's m_Frustum array. This array stores the planes in <a,b,c,d> format so we know that the normal is contained in members a, b, and c.

```
    // If the 'last outside plane' index was specified, test it first!
    if ( nLastOutside >= 0 && ( ((nBits >> nLastOutside) & 0x1) == 0x0 ) )
    {
        // Store the plane normal
        Normal = D3DXVECTOR3( m_Frustum[nLastOutside].a,
                              m_Frustum[nLastOutside].b,
                              m_Frustum[nLastOutside].c );
```

We then use the plane normal to calculate the near and far points on the bounding box with respect to the plane.

> **Note:** Remember from our Plane/AABB intersection discussion that the near point is the point that would intersect the plane first were it located totally in the plane's front space and slowly moved towards the plane until the point of intersection. The far point the last point on the AABB that would cross the plane in the same scenario.

```
        // Calculate near / far extreme points
        if ( Normal.x > 0.0f ) { FarPoint.x  = Max.x; NearPoint.x = Min.x; }
        else                   { FarPoint.x  = Min.x; NearPoint.x = Max.x; }

        if ( Normal.y > 0.0f ) { FarPoint.y  = Max.y; NearPoint.y = Min.y; }
        else                   { FarPoint.y  = Min.y; NearPoint.y = Max.y; }

        if ( Normal.z > 0.0f ) { FarPoint.z  = Max.z; NearPoint.z = Min.z; }
        else                   { FarPoint.z  = Min.z; NearPoint.z = Max.z; }
```

If the near point is found to be in front of the plane, the entire box must be in front of the plane. Consequently, if the box is totally in front of any of the planes, then it must be completely outside the frustum. As soon as such a plane is found, we return FRUSTUM_OUTSIDE

> **Note:** Our frustum planes normals face outwards.

```
        // If near extreme point is outside, then the AABB is totally outside
        if ( D3DXVec3Dot( &Normal, &NearPoint ) + m_Frustum[nLastOutside].d>0.0f )
           return CCamera::FRUSTUM_OUTSIDE;
```

If we have not returned from the function yet, it means the near point is in the backspace of the plane, so we next test to see if the far point is in the front space. If it is, the box is spanning the frustum plane and we return FRUSTUM_INTERSECT.

```
        // If far extreme point is outside, then the AABB is intersecting
        if ( D3DXVec3Dot( &Normal, &FarPoint ) + m_Frustum[nLastOutside].d > 0.0f )
            Result = CCamera::FRUSTUM_INTERSECT;
```

If we have still not returned from the function, it must mean that both the near and far points of the box are behind the frustum plane. This means the box is completely contained in the backspace of the plane and we must test the rest of the frustum planes. However, if the box is completely contained in the backspace of this plane, it also means all of the node's children will be too. Therefore, when this function is called for the child nodes, we should skip testing this plane and assume that they are in the backspace also. To do this, we set the bit for this plane in the nBits char to update the bit set that is passed down to the child nodes. We set the bit that corresponds to this plane by shifting the value 1 by the appropriate number of bits to the right (the number of bits to shift is equal to the index of the plane we are setting the bit for) and OR'ing it with the current bit set.

```
        else
            nBits |= (0x1 << nLastOutside); // We were inside, update our bit set
```

```
    } // End if last outside plane specified
```

If we reach this part of the function, it means that either no valid LastFrustumPlane value for the node was passed or that it was tested first, but the node is now found to be intersecting or behind that plane.

In the next section we simply create a loop from 0 to 6 to loop through the 6 frustum planes so that we can test the rest of them. We will skip a plane if its bit is already set to 1 in the nBits array and also skip the plane that was tested first in the above section of code (whose index is contained in nLastOutside).

```
    // Loop through all the planes
    for ( i = 0; i < 6; i++ )
    {
        // Check the bit in the uchar passed to see if it should be
        // tested (if it's 1, it's already passed)
        if ( ((nBits >> i) & 0x1) == 0x1 ) continue;

        // If 'last outside plane' index was specified,
        // skip if it matches the plane index
        if ( nLastOutside >= 0 && nLastOutside == (signed char)i ) continue;
```

If we get this far in the loop code, it means the current plane being processed has not yet had its bit set to 1 in the frustum bits char and it is not the plane we have already tested. As with the first plane we tested, we extract the normal of the current plane and calculate the near and far points.

```
        // Store the plane normal
        Normal = D3DXVECTOR3( m_Frustum[i].a, m_Frustum[i].b, m_Frustum[i].c );

        // Calculate near / far extreme points
        if ( Normal.x > 0.0f ) { FarPoint.x  = Max.x; NearPoint.x = Min.x; }
        else                   { FarPoint.x  = Min.x; NearPoint.x = Max.x; }

        if ( Normal.y > 0.0f ) { FarPoint.y  = Max.y; NearPoint.y = Min.y; }
        else                   { FarPoint.y  = Min.y; NearPoint.y = Max.y; }

        if ( Normal.z > 0.0f ) { FarPoint.z  = Max.z; NearPoint.z = Min.z; }
        else                   { FarPoint.z  = Min.z; NearPoint.z = Max.z; }
```

Next we calculate the distance from the near point to the plane. If it is found to be in the frontspace of the plane, we know the entire box must be outside the frustum so we can return FRUSTUM_OUTSIDE. However, before we return, we also store the index of this plane in the node's LastFrustumPlane member (pointed to by LastOutside) so that this function will test this plane first when the node is frustum tested again in the next visibility pass. We also copy the contents of the nBits char which currently contains all the frustum planes that the node is totally inside of. This is assigned to the FrustumBits pointer so that it is returned from the function to the node, where it can be passed down to its children.

```
        // If near extreme point is outside, then the AABB is totally outside
        // the frustum
        if ( D3DXVec3Dot( &Normal, &NearPoint ) + m_Frustum[i].d > 0.0f )
```

```
        {
            // Store the 'last outside' index
            if ( LastOutside ) *LastOutside = (signed char)i;

            // Store the frustm bits so far and return
            if (FrustumBits) *FrustumBits = nBits;
            return CCamera::FRUSTUM_OUTSIDE;

        } // End if outside frustum plane
```

If the near point is not in front of the plane but the far point is, it means the box is spanning the plane so we return FRUSTUM_INTERSECT.

```
        // If far extreme point is outside, then the AABB is intersecting
        // the frustum
        if ( D3DXVec3Dot( &Normal, &FarPoint ) + m_Frustum[i].d > 0.0f )
            Result = CCamera::FRUSTUM_INTERSECT;
```

Otherwise, it means the current plane being processed has the box contained totally in its backspace. Since this means that all of the node's children will also share this relationship with the plane, we set the bit that corresponds to this plane so that the children know they do not have to process it.

```
        else
            nBits |= (0x1 << i);
           // We were totally inside this frustum plane, update our bit set

    } // Next Plane
```

If we have not yet returned from the function it means the box is inside the frustum. Since there was no plane that rejected it, we set the node's last plane index (pointed to by LastOutside) to -1. We then copy the nBits array into the FrustumBits parameter so that they are accessible to the caller on function return. We then return a result of FRUSTUM_INSIDE.

```
    // Store none outside
    if ( LastOutside ) *LastOutside = -1;

    // Return the result
    if (FrustumBits) *FrustumBits = nBits;
    return Result;
}
```

We have now seen all of the code involved in the visibility processing procedure for the tree. This is all invoked when the application calls the ISpatialTree::ProcessVisibility function. In the last and final section covering the CBaseTree rendering system, we will examine how the visible data is rendered.

15.4.7 Rendering the Visible Static Polygon Data

After the application has called the ISpatialTree::ProcessVisibility method to flag the visible leaves and build the render batches in each leaf bin, all it has to do is loop through each subset in use by the scene and call the ISpatialTree::DrawSubset method. This method is shown below and is the final function of CBaseTree we will cover that pertains to its internal rendering system.

DrawSubset - CBaseTree

This method is called by the application to draw a particular subset in the tree. The application will typically set the texture and material required to render this subset prior to making this call. This function essentially just calls other functions which we have already covered.

Provided the device is valid, we use the CBaseTree::GetLeafBin method to fetch a pointer to the leaf bin for the associated subset ID. If a leaf bin does not exist for this subset ID, it means the tree contains no polygon data that uses this subset. In such situations, the GetLeafBin function returns NULL and we return from the function without taking any further action.

> **Note:** Since the scene is using global attribute IDs for all objects, it is entirely possible that the tree's polygon data will only use a handful of those attributes. As the application will essentially call DrawSubset for each global attribute, this function may be called many times with an attribute ID for which no polygon data exists in the tree for and for which no leaf bin has been created.

```
void CBaseTree::DrawSubset( unsigned long nAttribID )
{
    // Can draw?
    if ( !m_pD3DDevice ) return;

    // Retrieve the applicable leaf bin for this attribute
    CLeafBin * pLeafBin = GetLeafBin( nAttribID );
    if ( !pLeafBin ) return;

    // Render the leaf bin
    pLeafBin->Render( m_pD3DDevice );
}
```

After a valid leaf bin has been retrieved, its Render method is called. We looked at this function earlier and saw how it rendered all the render batches for each index/vertex buffer combination it contains triangles for.

15.5 Rendering the Tree – Application Perspective

Rendering the tree's static geometry could not be easier. Below we see the CScene::Render method from Lab Project 14.1. Because this function is now getting quite large (given all of the code to set up render states, enables lights and fog, rendering the CObject array, etc.), we have snipped a good amount of the code out to condense the listing.

You will recall that the first part of this function sets up certain render states, renders the sky box and enables lighting and fog. In this lab project we have added a new function call to the ISpatialTree::ProcessVisibility method at the bottom of the next section of code.

```
void CScene::Render( CCamera & Camera )
{
    ULONG i, j;
    long  MaterialIndex, TextureIndex;

    if ( !m_pD3DDevice ) return;

    //……SNIP : Set up Global Render states here

    // Render the skybox first !
    RenderSkyBox( Camera );

    //……SNIP : Set up lights and fog here……

    // Allow the spatial tree to process visibility
    m_pSpatialTree->ProcessVisibility( Camera );
```

Now that the visibility status of the tree has been updated using the camera's current position and orientation, we set the device world matrix to an identity matrix because the static geometry stored in the tree is already in world space.

```
    // Loop through each scene owned attribute
    D3DXMATRIX mtxIdentity;
    D3DXMatrixIdentity( &mtxIdentity );

    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxIdentity );
```

Now it is time to render the subsets of the spatial tree. We do this by looping through the array of attributes used by the scene. For each attribute, we set its texture and material on the device and call the ISpatialTree::DrawSubset method, passing in the respective attribute ID. If the tree has any data for this attribute, its associated leaf bin will render its triangles.

```
    for ( j = 0; j < m_nAttribCount; j++ )
    {
        // Retrieve indices
        MaterialIndex = m_pAttribCombo[j].MaterialIndex;
        TextureIndex  = m_pAttribCombo[j].TextureIndex;
```

```
        // Set the states
        if ( MaterialIndex >= 0 )
            m_pD3DDevice->SetMaterial( &m_pMaterialList[ MaterialIndex ] );
        else
            m_pD3DDevice->SetMaterial( &m_DefaultMaterial );

        if ( TextureIndex >= 0 && m_pTextureList[ TextureIndex ] )
            m_pD3DDevice->SetTexture( 0, m_pTextureList[ TextureIndex ]->Texture );
        else
            m_pD3DDevice->SetTexture( 0, NULL );

        // Render all faces with this attribute ID
        m_pSpatialTree->DrawSubset( j );

    } // Next Attribute
```

At this point the tree has been rendered in its entirety, so next we loop though the scene's CObject array (which will now store only the dynamic objects in use by the scene, like CActors) and render each one. We will discuss how this works in the next section. Then we can iterate through our terrain array and render any terrains in use. We do not show the code for this as it has not changed from previous demos (we will cover dynamic object rendering in the next section).

Finally, our CGameApp class now includes a new member called m_bDebugDraw. This is a Boolean that has its state toggled by a 'Debug Draw' menu option. The CGameApp::GetDebugDraw method returns the state of this Boolean. As you can see in the following code, if this function returns true, it means the user would like to enable debug drawing and thus the spatial tree's DebugDraw method is called. We discussed the code to this function in the previous lesson and saw how it traverses the tree and draws the bounding boxes at each node.

```
    // …… SNIP : Render all CObjects here (dynamic actors and tri mehses)

    // …… SNIP : Render any terrains here

    // Draw the spatial tree's debug data if requested
    if ( GetGameApp()->GetDebugDraw() ) m_pSpatialTree->DebugDraw( Camera );

}
```

And there we have it. That is all of the code involved in render our spatial tree from the application's perspective. All it took was two function calls to ISpatialTree member functions.

We have now covered the complete rendering system used by CBaseTree. In the end, we have built ourselves a very useful set of spatial managers that optimize both collision queries and scene rendering. In the final section of this chapter we will examine how our dynamic objects can also benefit from the visibility information in our tree. This will enable us to render only the actors that are currently considered visible by the tree.

15.6 Managing Dynamic Objects

So far we have implemented spatial trees that compile static polygon data and we have examined the methods used to prepare and render that data in an efficient manner. We have also added the concept of linking external objects to the leaves of the tree using the concept of detail areas. You will recall that a detail area is a bounding volume that helps shape the tree as it is being built. It is assigned during the compilation phase to the leaves in which it is fully or partially contained. Because each detail area compiled into the tree can also store a context pointer, they allow us to store any type of external object in the tree. A detail area may represent a series of render states, for example, that must be set on the device when the player enters the same leaves as the detail area. This might be handy if you wanted to use detail areas to represent regions of fog within the level or to represent an area where lights should be enabled/disabled. A detail area might also be used to represent a mesh or an actor that we do not want compiled into the tree at the per-polygon level, although we still want it to use the tree's visibility system to inform the application if that object should be rendered. The tree has no understanding of what a detail area represents internally. It just sees it as a bounding volume that has its pointer stored in the leaves in which it is contained. However, the application will know exactly what that detail area represents as it was responsible for registering it with the tree and assigning its context pointer. This context pointer will likely point to some structure or object that has meaning to the application.

The application is ultimately responsible for when and how to process detail areas. If we think about a detail area that represents an external mesh for example, the application would be responsible for rendering that mesh. However, if the detail area is not in any currently visible leaves, the application then knows it does not need to be rendered. In this situation the application can simply query the tree for a list of visible leaves and process only the detail areas in those leaves.

As useful as detail areas are, they do not provide us with coverage for all situations. Since they are compiled directly into the tree, they are totally static concepts. We can link an external mesh or actor to the tree at compile time by registering it as a detail area, but if the application plans on animating or updating the position of that object, detail areas will not suffice. Therefore, a system will have to be added to our tree that works in a similar manner to the detail area idea, but allows for an entity to have its position within the tree dynamically updated.

We will only need to add a handful of methods to CBaseTree in order for our trees to support dynamic objects. The system we will use to register dynamic objects with the tree will be similar to the system we used to register dynamic objects with the collision system. The ISpatialTree::InsertTreeObject method will be used for this purpose. It will store the object internally in the tree and return an ID (a handle) back to the caller. The caller can then use this handle to query for information about that object (such as what leaves is it currently in).

15.6.1 The TreeObject Structure

When a dynamic object is registered with the tree, a TreeObject structure is used as the transport mechanism to pass information about the object to the InsertTreeObject method. This structure is defined in ISpatialTree.h and is shown below.

Excerpt from ISpatialTree.h

```
typedef struct _TreeObject
{
    void          * pContext;
    bool          * pbVisible;
    long            nTreeObjectIndex;
} TreeObject;
```

When we pass a structure of this type into the InsertTreeObject method, only the first two members are used to describe the object. On function return, the third member (nTreeObjectIndex) will contain the handle assigned to the object by the spatial tree. The application can then store this returned handle in the object and use it for later querying.

void *pContext
This member is a pointer to some object that the application would like associated with the dynamic object in the tree. This could be a pointer to a CActor object for example. In our lab project code, we will pass a pointer to a CObject, which will contain either a dynamic CTriMesh or CActor object.

bool *pbVisible
This member allows the application to store a Boolean pointer that the tree will automatically update if the object is inside a visible leaf. That is, during the visibility pass, if a leaf is encountered that is visible, any tree objects (dynamic objects) that have been assigned to that leaf will have their Booleans set to true. This provides an alternative way for the application to query the visibility status of a dynamic object rather than fetching the visible leaves and searching them for the object currently being processed.

In Lab Project 14.1, we extend our CObject structure to store a Boolean member called m_bVisible. We will pass a pointer to this Boolean in this member of the TreeObject structure when we register the CObject with the spatial tree. Later, when the ProcessVisibility process is performed and a leaf which contains this object is found to be visible, the value of this Boolean will automatically be set to true by the tree even though it is stored in a CObject structure which has nothing to do with the tree. When rendering dynamic objects, the application can just loop through each one and only render it if the tree has set its Boolean visibility status to true (more on this later)

long nTreeObjectIndex
This is not an input parameter to ISpatialTree::InsertTreeObject; it will be set on function return so that the application can retrieve and store this value in the CObject structure. Just like our collision system, this is a unique ID that the tree has given to this pair of context and Boolean pointers inside its internal arrays.

15.6.2 The CObject Structure

Our CObject structure has been updated to include two new members. The new version of this structure is shown below with the new members highlighted in bold.

```
class CObject
{
public:

      // Constructors & Destructors for This Class.
         CObject( CTriMesh * pMesh  );
         CObject( CActor   * pActor );

             CObject( );
    virtual ~CObject( );


    // Public Variables for This Class
    D3DXMATRIX                  m_mtxWorld;
    CTriMesh                   *m_pMesh;
    CActor                     *m_pActor;
    LPD3DXANIMATIONCONTROLLER   m_pAnimController;
    CActionStatus              *m_pActionStatus;
    long                        m_nObjectSetIndex;

    long                        m_nTreeObjectIndex;
    bool                        m_bVisible;            // Is this object visible?
};
```

m_nTreeObjectIndex
This member will contain the handle (unique ID) assigned to the object by the spatial tree in response to the ISpatialTree::InsertTreeObject method being called for this object. Whenever we wish to update the position of the object or remove it from the spatial tree, it is this handle that we pass into the ISpatialTree::UpdateTreeObject and the ISpatialTree::RemoveTreeObject methods so that the tree knows exactly which tree object we are referring to.

m_bVisible
This member is a Boolean that describes the visibility status of the object. The address of this Boolean will be passed into the ISpatialTree::InsertTreeObject method (via the TreeObject structure) when the object is first registered. The spatial tree will automatically set this Boolean to true when the object is found to be in a visible leaf. The scene now has visibility information for the external object automatically with having to intersect with the tree itself. The tree will automatically set the m_bVisible Booleans of any CObjects in use by the application that have been registered with the spatial tree and are currently in visible leaves.

Before we discuss the ISpatialTree management of dynamic objects, we will take a look at how the application registers and updates the positions of dynamic objects using the ISpatialTree member functions. This will give us a better understanding of the system we need to implement and the way our application will expect that system to behave.

15.6.3 Registering a Dynamic Object with the Spatial Tree

As was the case with our collision system, any dynamic object that we create is registered with the spatial tree and returned a unique ID for that object within the system. In Lab Project 14.1 we load our scenes from IWF files and, as we have seen, the static geometry stored in the IWF file is added to the tree as static polygon data inside the CScene::ProcessVertices function. We have also seen in nearly all previous lab projects, that our dynamic objects are usually stored in the IWF file as external references to X files which are loaded into actors and stored in the scene's CObject array. It is the ProcessReference method that has always been the function used to create and load such X files into actors. It is in this function that the newly created CObject structure is registered with our collision system. It is also the function where we will register the object with the spatial tree.

The following snippet of code has been added to the very bottom of the CScene::ProcessReference function. It is responsible for adding the newly created CObject (which at this point may store a pointer to a CActor or a CTriMesh) with the spatial tree. In this code, which is only executed if the scene has a spatial tree, the new CObject which has been created earlier in the function (and populated accordingly) has been assigned to the pNewObject pointer.

```
// Add to the spatial tree's object list if applicable
if ( m_pSpatialTree )
{
  TreeObject Object;
  Object.pbVisible = &pNewObject->m_bVisible;
  Object.pContext  = NULL;

  // Add to the tree
  pNewObject->m_nTreeObjectIndex = m_pSpatialTree->InsertTreeObject( Object );

} // End if has spatial tree
```

As you can see, a new TreeObject structure is instantiated as our means of describing the dynamic object to the tree. The TreeObject's Boolean pointer is assigned to point at the m_bVisible Boolean member in our CObject structure for later updates during visibility testing. Notice how we set the context pointer of the TreeObject structure to NULL as we do not need to use it. You could assign this to point at the CObject structure itself, which would be useful if your means for determining dynamic object visibility was to loop through the currently visible leaves searching for TreeObjects that are then rendered. However, our current application only needs to know whether any of the CObjects in its array need to be rendered. Since we have registered the object's Boolean, this will be adequate for our purposes. All our application will need to know when looping through the CObject array is which ones need to be rendered. How you choose to do it is up to you as both methods suit different situations. Using the Boolean means that the application is not required to perform queries into the tree to determine visibility status.

After we have set up the TreeObject structure we then call ISpatialTree::InsertTreeObject passing in TreeObject structure so that this object can be stored in the tree's dynamic object array. The return value from this function is the handle (ID) of the new dynamic object we have just added which has been assigned by the spatial tree. We store this in the CObject's m_nTreeObjectIndex member so that when

our application alters the position of this object in the scene, it can pass this ID into the ISpatialTree::UpdateTreeObject to inform the tree that the object has been updated and that the leaves in which it is currently contained need to be recalculated.

Note that all we have done here is register a single Boolean pointer with the tree. We have not even assigned the object a bounding volume. So how can the tree know which leaves the object should be assigned to? The answer is that it does not; at least not currently. All the above function does is create an entry in the tree's dynamic object array where the passed Boolean pointer and context data pointer are stored. At this point, it is not assigned to any leaves. This is what the ISpatialTree::UpdateTreeObject method is for.

15.6.4 Updating Dynamic Tree Objects

When the application updates the position of an object (i.e., changes its world matrix), we must inform the tree so that it can remove that object's pointer from any leaves in which it is currently contained and assign it to the appropriate new leaves. This is all done automatically when the application makes a call to the ISpatialTree::UpdateTreeObject function. It is this function that is passed the ID of the object we would like to update and a world space AABB. This function will first remove the object from any leaves in which it is currently assigned. It will then pass the AABB of the object down the tree (using the CollectLeavesAABB method) to collect a list of leaves in which the bounding box is contained. The object then has its pointer added to each leaf in which it is contained. This same list of leaves is then stored inside the tree in a structure that pairs the tree object with its intersected leaf list.

Each leaf in our tree will now potentially store a list of TreeObject pointers and each TreeObject stored in the tree will also be stored alongside a list of leaves in which the object is currently contained. This provides yet more convenient ways to use dynamic objects with the system. As each tree object is stored in an array paired with a list of leaves in which it is currently contained, this means the application can quickly access the leaf list for an object using the ISpatialTree::GetTreeObjectLeaves method. This method is passed the ID of a tree object and will simply look up that object in the tree's internal array and return the leaf list that is stored alongside it. Additionally, because the leaves themselves also store a list of tree object pointers describing the list of dynamic objects contained within them, the application can call the ISpatialTree::GetVisibleLeafList method and then parse the returned visible leaves to see if the object in which it is interested is stored there. These few methods provide the application with choices about the way it would like to work with and query the status of its dynamic objects.

The following section of code is taken from the CScene::AnimateObjects method in Lab Project 14.1. This function is certainly familiar to us at this point, but now a few new lines have been added so that the spatial tree is informed about object position updates so that it can rebuild the leaf lists for that object.

The first section of the function is unchanged from our previous lab projects. It sets up a loop to iterate though every object in the scene's CObject array and fetches the CActor and CTriMesh pointers into

local variables. If the actor pointer is valid, we attach the object's controller to the actor and call the actor's AdvanceTime method to advance the timeline of any animation it may be playing.

```
void CScene::AnimateObjects( CTimer & Timer )
{
    ULONG i;

    // Process each object for coll det
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CObject * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Get the pointers
        CActor   * pActor = pObject->m_pActor;
        CTriMesh * pMesh  = pObject->m_pMesh;
        if ( !pActor && !pMesh ) continue;

        // Update actor?
        if ( pActor )
        {
            if ( pObject->m_pAnimController )
                 pActor->AttachController( pObject->m_pAnimController,
                                           false,
                                            pObject->m_pActionStatus );

            // Advance time
            pActor->AdvanceTime( Timer.GetTimeElapsed(), false );

        } // End if actor
```

The next section of code is only executed is the current object being processed has been registered with the collision system or the spatial tree. If so, then we will set the actor's matrix and force its absolute matrices to be updated. This is important because we shall see later that this is used to calculate the world space bounding box for the actor which we will need in order to update its position in the tree.

```
        if ( pObject->m_nObjectSetIndex > -1 || pObject->m_nTreeObjectIndex > -1 )
        {
            // Set world matrix and update combined frame matrices.
            if ( pActor ) pActor->SetWorldMatrix( &pObject->m_mtxWorld, true );
```

If the CObject has a valid collision system handle, we call the collision system's ObjectSetUpdated method to allow it to recalculate information about its dynamic object array.

```
            // Notify the collision system that this set of dynamic objects
            // positions, orientations or scale have been updated.
            if ( pObject->m_nObjectSetIndex > -1 )
                 m_Collision.ObjectSetUpdated( pObject->m_nObjectSetIndex );
```

The next bit is new. If the application is using a spatial tree and the current CObject being processed has a valid spatial tree handle stored in its m_nTreeObjectIndex structure, it means that this object has been

registered with the tree and therefore (as its position may have been updated by the AdvanceTime call) we should inform the spatial tree to recalculate the leaves in which this object is situated.

To instruct the spatial tree that the position of our object might have changed, we have to send the object's spatial ID and its world space bounding box into the ISpatialTree::UpdateTreeObject method. Until now, we have had no methods exposed from either CTriMesh or CActor that return the world space bounding box; we will add them to these classes at the end of this lesson. For now, let us just assume that both CTriMesh and CActor have a method called GetBoundingBox which is passed two vectors that will be filled with the world space extents of the object's bounding box.

```
            // Update the tree information
            if ( m_pSpatialTree && pObject->m_nTreeObjectIndex > -1 )
            {
                D3DXVECTOR3 vecMin, vecMax;

                // Retrieve the bounding box
                if ( pActor )
                    pActor->GetBoundingBox( vecMin, vecMax );
                else
                    pMesh->GetBoundingBox( vecMin, vecMax, &pObject->m_mtxWorld );
```

The CActor::GetBoundingBox method takes only two output parameters for the purposes of retrieving the world space box extents that encompass the entire actor hierarchy. The CTriMesh object (which may alternatively be stored in a CObject) has no way of returning a world space bounding box unassisted. It is typically an object space mesh and has no knowledge of its world space position. Therefore, the CTriMesh object will store a model space bounding box which can be converted into world space by passing a transformation matrix as the third parameter to its GetBoundingBox function. This function will simply use our handy TransformAABB method to convert the mesh's model space AABB into a world space AABB. The extents will then be returned using the first two output parameters.

At this point, whether the CObject contains a mesh or an actor, we have a world space bounding box which we can pass into the tree.

```
                // Update the spatial tree object
                m_pSpatialTree->UpdateTreeObject( pObject->m_nTreeObjectIndex,
                                                  vecMin,
                                                  vecMax );

            } // End if has object index

        } // End if actor exists

    } // Next Object
}
```

As you can see, we pass the UpdateTreeObject method the bounding box of the object and its spatial ID, which was issued at the time of registration. Later in the lesson we will see exactly how the bounding boxes are calculated for meshes and actors and how all these new spatial tree methods work.

15.6.5 Rendering Dynamic Objects

In the following code we see a section of the CScene::Render method. Since this is a large method we will only show the new additions which pertain to the rendering of the scene's objects. Remember that they have all been registered with the spatial tree as dynamic objects,

As we discussed in the previous section, situated near the top of the CScene::Render function, the ISpatialTree::ProcessVisibility function is called to traverse the tree and set the visible leaves based on the input camera's frustum. Earlier in the lesson we also learned that this function also builds the render batches for the static data stored in each leaf bin. We will see in a moment that this method can be updated to handle dynamic objects as well. When any leaf is found to be visible, any TreeObject structures stored in that leaf will have their Boolean pointers set to true. We know that the Boolean pointer in each TreeObject structure is actually a pointer to the CObject::m_bVisible member variable, so when the ProcessVisibility method returns, any CObjects that are in visible leaves will have their m_bVisible members set to true.

In the next section of code we show the call to the ProcessVisibility method followed by the loop that iterates through the CScene's CObject array and renders any that have had their visibility Booleans set to true by the spatial tree.

```
    // Update the leaf visibility  (sets CObject Booleans)
    m_pSpatailTree->ProcessVisibility( pCamera);

     … SNIP : Render spatial tree's static data here

    // Process each object
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        CObject  * pObject = m_pObject[i];
        if ( !pObject ) continue;

        // Skip if this object is listed as not visible
        if ( !pObject->m_bVisible ) continue;

        // Flag this object as *not* visible for the next call to ProcessVisibility
        // but only if it has been registered with the tree
        if ( pObject->m_nTreeObjectIndex >= 0 ) pObject->m_bVisible = false;

        // Retrieve actor and mesh pointers
        CActor   * pActor = pObject->m_pActor;
        CTriMesh * pMesh  = pObject->m_pMesh;
        if ( !pMesh && !pActor ) continue;

        … SNIP : Set Object matrix
        …
        … For Each  Subset
        …
        …    pActor/pMesh -> DrawSubset ( … )
        …
        … End For Each Subset
```

```
    }

    // SNIP : Render any terrains here
```

The important point to notice in the above code snippet is that once we find a CObject that is in a visible leaf, we reset its visibility Boolean back to false prior to rendering it. This is so the Boolean will be returned to its default state when the ISpatialTree::ProcessVisibility method is called during the next frame render.

Now that we have had a high level look at how the spatial tree's dynamic object system will work, it should be clear that, from the application's perspective, things could not be easier. The application really only ever has to call two methods for each dynamic object. The InsertTreeObject method is called once (at load time in our example) to register the object with the tree, and the UpdateTreeObject method is called to update the position of the object inside the spatial tree when required. Let us now have a look at the changes we have had to make to both the ISpatialTree and ILeaf interfaces in order to add support for these concepts.

15.6.6 ILeaf - Adding Dynamic Object Support

The ILeaf abstract base class from which CBaseLeaf is derived only requires one extra method to be implemented in the derived classes so that application can query the list of TreeObjects for a given leaf. This method is called GetTreeObjectList and is highlighted in bold below.

Excerpt from ISpatialTree.h

```
class ILeaf
{
public:

    // Typedefs, Structures and Enumerators.
    typedef std::list<TreeObject*> TreeObjectList;


    // Constructors & Destructors for This Class.
    virtual ~ILeaf() {};     // forces derived classes to have virtual destructors


    // Public Pure Virtual Functions for This Class.
    virtual bool            IsVisible           ( ) const = 0;
    virtual unsigned long   GetPolygonCount     ( ) const = 0;
    virtual CPolygon *      GetPolygon          ( unsigned long nIndex ) = 0;
    virtual unsigned long   GetDetailAreaCount  ( ) const = 0;
    virtual TreeDetailArea *GetDetailArea       ( unsigned long nIndex ) = 0;

    virtual TreeObjectList& GetTreeObjectList   ( ) = 0;

    virtual void            GetBoundingBox      ( D3DXVECTOR3 & Min,
                                                  D3DXVECTOR3 & Max ) const = 0;

};
```

Notice that this method accepts no parameters and will return a list of all the TreeObject structures stored in this leaf. The return type TreeObjectList is a type definition for an STL list which stores TreeObject structure pointers. You can see this typedef at the top of the code shown above.

As you can see, from the application's perspective (which always works with the ISpatialTree interface), only one method has been added that allows it to retrieve the entire list of dynamic objects stored in that leaf. An application might, for example, employ a totally different rendering strategy than the one we have chosen. It might prefer to fetch the list of visible leaves from the tree and then loop through each of them individually. For each leaf, it can use this new method (GetTreeObjectList) to get the list of dynamic objects stored in that leaf and then fetch the context pointer from each TreeObject structure for further processing. This is an alternative to the Boolean pointer method we have chosen to use in Lab Project 14.1.

15.6.7 ISpatialTree - Adding Dynamic Object Support

Four new methods have been added to the ISpatialTree interface to allow the application to add, insert, and remove dynamic objects from the tree. There is also a method that allows the application to retrieve a list of all the leaves a given object is currently contained in.

We will not show the entire ISpatialTree class here since we have only added four methods. These are shown below.

Excerpt from ISpatialTree.h

```
virtual long            InsertTreeObject    ( TreeObject & Object ) = 0;

virtual void            UpdateTreeObject    ( long nObjectIndex,
                                              const D3DXVECTOR3 &BoundsMin,
                                              const D3DXVECTOR3 &BoundsMax)= 0;

virtual void            RemoveTreeObject    ( long nObjectIndex ) = 0;

virtual bool            GetTreeObjectLeaves ( long nObjectIndex,
                                              LeafList & List ) = 0;
```

These are all pure virtual methods which serve to define the functionality that must be implemented in the derived classes.

15.6.8 CBaseTree/CBaseLeaf - Adding Dynamic Object Support

The full implementation for dynamic object support will be contained within CBaseTree and CBaseLeaf. The derived classes will not require any additional code. We will start by examining the members and methods that will need to be added to CBaseLeaf first.

15.6.9 CBaseLeaf – The Source Code

CBaseLeaf will need three new member functions and a single member variable. In the following listing we do not show the entire declaration of CBaseLeaf, only the new members.

```
public:

    // Public Virtual Functions for This Class (from base).
    virtual TreeObjectList& GetTreeObjectList   ( );


    // Public Functions for This Class.
    void                    InsertTreeObject    ( TreeObject * pObject );
    void                    RemoveTreeObject    ( TreeObject * pObject );

protected:

    // Protected Variables for This Class
    TreeObjectList      m_TreeObjects;      // List of objects
```

m_TreeObjects
This member will be used to store a list of TreeObject structures. This list represents all the tree objects that are currently considered to be in this leaf by the CBaseTree. A dynamic object (a TreeObject) will be assigned to a leaf (or multiple leaves) when the application issues a call to the CBaseTree::UpdateTreeObject method. This method will traverse the tree with an input world space bounding box and collect a list of all leaves intersecting that box. The CBaseLeaf::InsertTreeObject method will then be called for each leaf in this list so that the tree object is added to each leaf's tree object list.

InsertTreeObject – CBaseLeaf

This method will never be called by the application. It is used during a tree object update by CBaseTree when it determines that the tree object should be added to a leaf. The tree will issue a call to this method for each leaf in which the tree object is contained. This method is a simple function that just adds the passed TreeObject pointer to the leaf's internal list of tree objects.

```
void CBaseLeaf::InsertTreeObject( TreeObject * pObject )
{
    // Push this context pointer onto the end of the list
    m_TreeObjects.push_back( pObject );
}
```

RemoveTreeObject – CBaseLeaf

This is another method that will never be called by the application, but is called by the tree during the update of a tree object. When the application issues a call to the CBaseTree::UpdateTreeObject method, the first thing this method will do is remove the tree object from any leaves to which it is currently assigned. We will see in a moment that, inside CBaseTree, each tree object is stored along with a list of leaves in which it is currently contained. Therefore, when its position needs to be updated, this leaf list is traversed and the CBaseLeaf::RemoveTreeObject method will be called to unhook the tree object from all its current leaves. The update method will then pass the bounding box of the tree object (in its new world space position) down the tree and collect a new list of leaves in which the object is now considered to be contained. This new leaf list is then traversed and the CBaseTree::InsertTreeObject method called for each. This will add the tree object to each of the new leaves as discussed above. Therefore, updating a tree object will essentially involve removing it from all its current leaves, finding a new list of leaves its bounding box intersects, and then adding the object to this new list of leaves. The code to CBaseLeaf::RemoveTreeObject, which is used in this process, is shown below.

```
void CBaseLeaf::RemoveTreeObject( TreeObject * pObject )
{
    // Remove this context pointer from the list
    m_TreeObjects.remove( pObject );
}
```

As you can see, it simply removes the passed TreeObject pointer from the leaf's tree object list. At that point, the tree object will no longer be considered to be in that leaf.

GetTreeObjectList - CBaseLeaf

This method is required by the base class (ILeaf) and allows the application to retrieve the list of tree objects currently contained in the leaf. This method just returns the leaf's m_TreeObjects list.

```
ILeaf::TreeObjectList& CBaseLeaf::GetTreeObjectList( )
{
    // Just return the list.
    return m_TreeObjects;
}
```

As mentioned, this might prove useful if the application chooses a different rendering strategy than the one we are using.

SetVisible – CBaseLeaf

We have examined this function a few times in this lesson in one form or another. You will recall that it is called by a derived class's UpdateTreeVisibility method every time a visible leaf is found during the visibility traversal. It is called to flag leaves as either invisible or visible.

As we know, this function sets the visible status of the leaf. If the leaf is visible, its triangle runs are added to the relevant leaf bins. After that, the leaf then adds its 'this' pointer to tree's visible leaf list. This is all unchanged from the previous version. All we have added to the bottom of the function is a couple of lines that loop through the list of tree objects stored in this leaf and sets their registered Booleans to true. The new lines of code are highlighted in bold at the bottom of the function listing.

```
void CBaseLeaf::SetVisible( bool bVisible )
{
    ULONG                 i, j;
    RenderData::Element * pElement;
    CLeafBin            * pLeafBin;
    RenderData          * pData;

    // Flag this as visible
    m_bVisible = bVisible;

    // If we're being marked as visible, inform the renderer
    if ( m_bVisible && m_nRenderDataCount > 0 )
    {
        // Loop through each renderable set in this leaf.
        for ( i = 0; i < m_nRenderDataCount; ++i )
        {
            pData    = &m_pRenderData[i];
            pLeafBin = pData->pLeafBin;

            // Loop through each element to render
            for ( j = 0; j < pData->ElementCount; ++j )
            {
                pElement = &pData->pElements[j];
                if ( pElement->PrimitiveCount == 0 ) continue;

                // Add this to the leaf bin
                pLeafBin->AddVisibleData( pElement->VBIndex,
                                          pElement->IndexStart,
                                          pElement->PrimitiveCount );
            } // Next Element
        } // Next RenderData Item
    } //  End if visible


    // Update tree object's if we're visible
    if ( m_bVisible )
    {
        // // Add this leaf to the tree's visible leaf list
        m_pTree->AddVisibleLeaf( this );
```

```
        TreeObjectList::iterator Iterator = m_TreeObjects.begin();
        for ( ; Iterator != m_TreeObjects.end(); ++Iterator )
        {
            TreeObject * pObject = *Iterator;
            if ( pObject->pbVisible ) *pObject->pbVisible = true;

        } // Next tree object

    } // End if we are visible
}
```

As you can see, we iterate through the TreeObject list stored in this leaf only if the leaf is visible. For each tree object, if it has a non-NULL Boolean pointer (pbVisible), then it means that the application would like this Boolean set to true when the object is visible. As we saw earlier, in our application, the Boolean pointer in each tree object will actually point to a CObject's visibility Boolean. This means that although the tree itself has no knowledge that this tree object actually represents a CObject structure, it still has the ability to set its visibility status to true so that the application knows it has to render this CObject during the CScene::Render function.

We have now looked at the minor changes to CBaseLeaf that provide support for the containment of dynamic objects. Next we will discuss the changes to the CBaseTree class where most of our dynamic object support functionality will be contained.

15.6.10 CBaseTree – The Source Code

We have seen that our leaves will maintain a list of TreeObject structures, so every leaf will know which dynamic objects it contains. However, CBaseTree will also store an array (STL vector) of all TreeObjects currently registered with the system, along with a list of leaves that object is currently contained within. This means that each leaf will have immediate access to the objects it contains, and each object will have immediate access to the list of leaves in which it is contained. This allows us to very efficiently return a list of leaves when the application issues a call to the ISpatialTree::GetTreeObjectLeaves method. This method is passed the ID of a tree object and returns its leaf list. The leaf list for each tree object is updated during the call to the CBaseTree::UpdateObject method, which we will see the code for in a moment.

15.6.11 The TreeObjectData Structure

A new structure will be needed so that CBaseTree can maintain a list of TreeObject structures paired with the object leaf lists. This new structure is defined in the CBaseTree namespace but is shown below on its own. It is called TreeObjectData and there will be an array of these structures stored in CBaseTree. Each element in this array will store the information for a tree object that is currently registered with the system.

Excerpt from CBaseTree.h

```
struct TreeObjectData             // Stores the data relating to a tree object
{
        TreeObject   Object;      // The 'context' for the tree object
        LeafList     Leaves;      // List of leaves in which the object exists
        bool         bInUse;      // This element is currently in use?
}
```

TreeObject Object

This member stores the TreeObject structure itself. This is the TreeObject that this TreeObjectData structure represents. Recall that the TreeObject structure has three members: a spatial ID that identifies the object to the system, a Boolean pointer that will be set to true when the object is visible, and a context pointer that can be used by the application to store any arbitrary data.

LeafList Leaves

This is a list of leaves in which the above TreeObject is currently considered contained. This leaf list is used to update the position of a tree object very efficiently. Because we store the list of leaves the object is currently contained in, when we wish to update the position of a tree object (which first involves removing it from all current leaves), we can iterate through this list calling the CBaseLeaf::RemoveTreeObject method before emptying the list. This will allow us to quickly empty this list and remove the object from all the leaves prior to an update. Then, the CBaseTree::UpdateTreeObject method will send the AABB of the tree object down the tree to build a new leaf list which is then stored in this member. We can then iterate through this list and call the CBaseLeaf::InsertTreeObject method for each leaf to add the TreeObject to each new leaf in which it is now contained.

Of course, the other useful thing about this member, beyond speeding up the tree object updates, is that it allows the application to retrieve a list of leaves for a given tree object. As this information is already compiled for every tree object, when the application requests the leaf list for a tree object by passing its spatial ID, the CBaseTree::GetTreeObjectLeaves method can simply search the TreeObjectData array for a TreeObject data structure with the matching ID and, if found, return the leaf list stored there.

bInUse

This is a Boolean that will tell the tree whether this particular TreeObjectData structure currently contains a tree object or whether it is empty and can be reused. This minimizes array resizing every time a tree object is removed. When a tree object is removed from the system (at the application's request), we do not delete its TreeObjectData structure and resize the array; we simply set this Boolean to false so that we know the next time we wish to add a new object, this TreeObjectData element can be used and its current data overwritten.

Below we see the new members and methods in CBaseTree that have been added to manage dynamic objects. As you can see, it implements the four methods required by ISpatialTree to allow the application to add, remove, and update dynamic objects within the tree. It contains only one new member, which is an array of TreeObjectData structures.

Excerpt from CBaseTree.h

```
public:

    virtual long            InsertTreeObject    ( TreeObject & Object );
    virtual void            UpdateTreeObject    ( long nObjectIndex,
                                                  const D3DXVECTOR3 & BoundsMin,
                                                  const D3DXVECTOR3 & BoundsMax );

    virtual void            RemoveTreeObject    ( long nObjectIndex );

    virtual bool            GetTreeObjectLeaves ( long nObjectIndex,
                                                  LeafList & List );

protected:

    // STL Typedefs
    typedef std::vector<TreeObjectData>     TreeObjectVector;

    // Protected Variables for This Class.
    TreeObjectVector            m_TreeObjects;
};
```

TreeObjectVector m_TreeObjects
This is an STL vector that stores TreeObjectData structures. Therefore, this array is basically used to store all of the tree's currently registered dynamic objects.

InsertTreeObject – CBaseTree

This is the method that the application uses to register a dynamic object with the system. The caller fills out a TreeObject structure and passes it in and the function will look for a place to store it in the TreeObjectData array. It first searches though the m_TreeObjects vector to see if there are any TreeObjectData structures that are not currently in use by the system. If one if found, we break from the loop so that the loop variable *i* contains the index of this element.

```
long CBaseTree::InsertTreeObject( TreeObject & Object )
{
    ULONG i;

    // Loop through the vector and determine if there is a free slot
    for ( i = 0; i < m_TreeObjects.size(); ++i )
    {
        // Is this in use?
        if ( m_TreeObjects[i].bInUse == false ) break;

    } // Next Tree Object
```

If we could not find a free slot, then the loop would have run to completion. Either way, loop variable *i* will now contain the new index for where we wish to store the passed tree object. We store this index in the object's nTreeObjectIndex member so that on function return the application will have access to this

ID and can store it. This is a unique ID for this object within the system and the handle the application will use to refer to it when it wants to update its position within the tree.

```
    // Update the object's index to the new slot
    Object.nTreeObjectIndex = i;
```

If *i* equals the current size of the array, then it means that there was no free slot that can be reused, and we will need to instantiate a new TreeObjectData structure, store the passed TreeObject in it, and set its bInUse Boolean to true. We then add it to the end of the array.

```
    // Did we reach the end?
    if ( i == m_TreeObjects.size() )
    {
        TreeObjectData Data;

        // Populate a new data element
        Data.bInUse   = true;
        Data.Object   = Object;

        // Add a new element to the array
        m_TreeObjects.push_back( Data );

    } // End if no free slot
```

If we did find a free slot in the array, we will reuse it by storing the passed TreeObject in it and setting its bInUse Boolean to true. Notice in the following code that because this TreeObjectData structure was used previously by another object that has since be un-registered, the leaf list stored there will need to be emptied as it will not be valid for our new object.

```
    else
    {
        // Just re-use the free slot
        m_TreeObjects[i].bInUse                 = true;
        m_TreeObjects[i].Object                 = Object;
        m_TreeObjects[i].Leaves.clear();


    } // End if free slot found

    // Return the new object's index
    return i;
}
```

Finally, the function returns the ID that was assigned to the new tree object. This ID is just the position of the TreeObjectData structure in the tree's object list.

What is important to remember is that although this function is used to add a new dynamic object to the tree, when the function returns it will have not yet be assigned to any leaves. That is what the UpdateTreeObject method does, which we will examine next.

UpdateTreeObject – CBaseTree

We saw earlier that this function is called by the application in the CScene::AnimateObjects method. It should be called whenever the world matrix of an object has been changed, or in the case of an animated actor, whenever its animation has been advanced. In the case of an animated actor, even if the application never changes its position in the scene, the individual meshes contained in its hierarchy may move or rotate. This would ultimately change the size of the actor's bounding box (the box that bounds all meshes in the actor) and if the bounding box has gotten bigger or smaller with respect to the previous frame update, the actor may now exist in more or less leaves than before.

The function is actually quite straightforward even though it would seem to have a difficult task to perform. It is passed the ID identifying the object that needs its leaves recalculated, and the world space bounding box of that object. The function uses the passed ID to fetch a pointer to the relevant TreeObjectData structure to update.

```
void CBaseTree::UpdateTreeObject( long nObjectIndex,
                                  const D3DXVECTOR3 & BoundsMin,
                                  const D3DXVECTOR3 & BoundsMax )
{
    LeafList::iterator Iterator;
    TreeObjectData *   pData = NULL;

    // Valid the specified data item.
    if ( nObjectIndex < 0 || nObjectIndex >= (signed)m_TreeObjects.size() ) return;
    if ( !m_TreeObjects[ nObjectIndex ].bInUse ) return;

    // Store pointer to object to save lookups
    pData = &m_TreeObjects[ nObjectIndex ];
```

As the object may have moved, we will first remove it from all leaves in which it is currently stored. As each TreeObjectData structure contains the TreeObject and its current leaf list, we just have to loop through that leaf list calling the CBaseLeaf::RemoveTreeObject method for each.

```
    // Loop through the leaves and remove this from the object list
    for ( Iterator = pData->Leaves.begin();
          Iterator != pData->Leaves.end(); ++Iterator )
    {
        CBaseLeaf * pLeaf = (CBaseLeaf*)(*Iterator);
        if ( !pLeaf ) continue;

        // Request that this object is removed from the leaf
        pLeaf->RemoveTreeObject( &pData->Object );

    } // Next Leaf

    // Clear the list
    pData->Leaves.clear();
```

We saw the code to the CBaseLeaf::RemoveTreeObject function a moment ago. It simply removes the passed TreeObject pointer from its list. After this loop ends, we will have visited every leaf that currently contains the object and removed its pointer from those leaves. After that, we empty the leaf list in the TreeObject as well. At this point our object has the same status as a newly registered object; it is not stored in any leaves and it does not have any leaves in its leaf list.

Now it is time to rebuild this information. We start by passing the TreeObject's empty leaf list into the CollectLeavesAABB method, along with the world space bounding box of the object. When this function returns, the object will have an updated leaf list.

```
    // Collect the new leaves
    CollectLeavesAABB( pData->Leaves, BoundsMin, BoundsMax );
```

Of course, our job is not quite done. Although the object now has an updated list of leaves, the leaves in this list still do not know that they contain the object. Therefore, we will loop through the new leaf list and call the CBaseLeaf::InsertTreeObject method to add the object to the object lists for each leaf in which it is currently contained.

```
    // Loop through the leaves and add this back to the newly discovered leaves
    for ( Iterator = pData->Leaves.begin();
          Iterator != pData->Leaves.end(); ++Iterator )
    {
        CBaseLeaf * pLeaf = (CBaseLeaf*)(*Iterator);
        if ( !pLeaf ) continue;

        // Request that this object is removed from the leaf
        pLeaf->InsertTreeObject( &pData->Object );

    } // Next Leaf
}
```

GetTreeObjectLeaves - CBaseTree

This function can be used by the application to fetch the list of leaves (both visible and invisible) in which a tree object is currently contained. The application passes in the ID of the tree object it would like to retrieve a leaf list for and a leaf list to store the results.

```
bool CBaseTree::GetTreeObjectLeaves( long nObjectIndex, LeafList & List )
{
    // Valid the specified data item.
    if ( nObjectIndex < 0 || nObjectIndex >= (signed)m_TreeObjects.size() )
         return false;

    if ( !m_TreeObjects[ nObjectIndex ].bInUse ) return false;

    // Populate the list
    List = m_TreeObjects[ nObjectIndex ].Leaves;
```

```
    // Success!
    return true;
}
```

This method uses the passed ID to fetch the TreeObjectData structure from the array and, provided this element is in use (i.e., it is a valid object), it returns the TreeObjectData's leaf list.

RemoveTreeObject - CBaseTree

This method can be called by the application to remove a dynamic object from the tree. It is the mirror function to the InsertTreeObject method. Its single parameter is the ID of the tree object you would like to unregister from the system.

The first thing the method does is use the passed ID to fetch the corresponding TreeObjectData structure from the tree's object data array.

```
void CBaseTree::RemoveTreeObject( long nObjectIndex )
{
    LeafList::iterator Iterator;
    TreeObjectData *   pData = NULL;

    // Valid index?
    if ( nObjectIndex < 0 || nObjectIndex >= (signed)m_TreeObjects.size() ) return;

    // Store pointer to object to save lookups
    pData = &m_TreeObjects[ nObjectIndex ];
```

Now we loop through this tree object's leaf list and call the CBaseLeaf::RemoveTreeObject method for each one. This removes it from the object list in each leaf currently containing the object.

```
    // Remove it from any leaves it currently exists in
    for ( Iterator = pData->Leaves.begin();
          Iterator != pData->Leaves.end(); ++Iterator )
    {
        CBaseLeaf * pLeaf = (CBaseLeaf*)(*Iterator);
        if ( !pLeaf ) continue;

        // Request that this object is removed from the leaf
        pLeaf->RemoveTreeObject( &pData->Object );

    } // Next Leaf
```

Finally, we set this TreeObjectData structure's bInUse Boolean to false and empty its leaf list.

```
    // Just re-use the free slot next time someone inserts
    pData->bInUse = false;
    pData->Leaves.clear();
}
```

We have now covered all the new methods and code that are involved in adding dynamic object support to CBaseTree.

15.6.12 Conclusion – Dynamic Objects

After an admittedly arduous journey, we have completed the code for our ISpatialTree derived classes. Along the way we have seen that CBaseTree provides nearly all of the core functionality for any tree type we might care to derive from it in the future.

One interesting item we encountered in this last section was that CBaseTree views dynamic objects in a very abstract way. It has no idea what the TreeObject structure represents and does not need to. We do not even use the context pointer of this structure in our lab project, only its Boolean pointer. Therefore, we have discovered that a dynamic object used in this way is really just a Boolean pointer with an assigned ID. It is ultimately attached to some number of leaves every time the UpdateTreeObject method is called, but it remains quite a separate concept. The TreeObject does not even store a bounding volume for the object it represents, as one might suspect. Instead, this bounding volume is passed into the UpdateTreeObject method by the application whenever it wishes to inform the tree that the object has moved and its Boolean pointer should now be assigned to a new set of leaves.

Of course, the dynamic object system we have developed does place a particular burden on the application. In order for the application to update the position of a CObject, it must have access to the world space bounding box of the object it contains (either a CTriMesh or a CActor). Up until this point in the course however, CTriMesh and CActor have never exposed a method for retrieving any world space bounding box, so we will need to add these now before concluding the lesson.

15.7 CTriMesh Revisited – World Space Bounding Boxes

CTriMesh will now have two new members added to it. These will be 3D vectors that describe the *object space* AABB of the mesh. As a mesh is by its very nature an object space concept, there is no way it can know about its world space position or size and therefore, the object space box will be created and stored instead. CTriMesh will then expose a method called GetBoundingBox which allows for a world transformation matrix to be passed. This function will transform the object space bounding box into world space before returning it to the application. We saw CTriMesh::GetBoundingBox being used earlier when we examined the changes to the CScene::AnimateObjects method. This method would fetch the world space bounding box from the CTriMesh (if one was stored in the current CObject structure being updated) and pass it into the ISpatialTree::UpdateTreeObject. When the CTriMesh::GetBoundingBox method was called, it was passed the world matrix of the owner object. This meant we would get back the world space AABB for the mesh.

Here are the new members and methods added to CTriMesh.

```
protected:
D3DXVECTOR3         m_vecBoundsMin;
D3DXVECTOR3         m_vecBoundsMax;

public:

HRESULT UpdateBoundingBox();

void    GetBoundingBox(D3DXVECTOR3 &BoundsMin,
                       D3DXVECTOR3 &BoundsMax,
                       D3DXMATRIX * pMatrix = NULL ) const;
```

UpdateBoundingBox - CTriMesh

Our application will never call this function (although it can if it would like to rebuild the object space bounding box of the mesh for some reason) as it is called automatically by the CTriMesh::LoadMeshFromX function just before returning. The function is like many we have seen before. The first thing it does is query the D3DXMesh managed by the CTriMesh object for an ID3DXBaseMesh interface. This allows us to work with both regular and progressive meshes using the base interface.

```
HRESULT CTriMesh::UpdateBoundingBox( )
{
    HRESULT           hRet;
    UCHAR           * pVertices  = NULL;
    ULONG             nVertexStride, nVertexCount;
    D3DXVECTOR3       vecPos;
    ULONG             i;

    // Retrieve the mesh
    LPD3DXBASEMESH pMesh = NULL;

    // What type of mesh?
    if ( m_pMesh )
    {
        // Query the interface to get back the base mesh.
        hRet = m_pMesh->QueryInterface( IID_ID3DXBaseMesh, (void**)&pMesh );
        if ( FAILED( hRet ) ) return hRet;

    } // End if standard mesh
    else if ( m_pPMesh )
    {
        // Query the interface to get back the progressive mesh
        hRet = m_pPMesh->QueryInterface( IID_ID3DXBaseMesh, (void**)&pMesh );
        if ( FAILED( hRet ) ) return hRet;

    } // End if progressive mesh
    else
    {
        // Just return
```

```
        return D3D_OK;

    } // End if no mesh
```

We initialize the mesh's bounding box to extreme starting values (an inside-out box).

```
    // Reset the bounding box
    m_vecBoundsMin = D3DXVECTOR3(  FLT_MAX,  FLT_MAX,  FLT_MAX );
    m_vecBoundsMax = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX );
```

We then get the number of vertices and the number of bytes each vertex consumes in memory before locking the mesh's vertex buffer to get a pointer to its vertex data.

```
    // Retrieve additional info we need for processing
    nVertexStride = pMesh->GetNumBytesPerVertex( );
    nVertexCount  = pMesh->GetNumVertices( );

    // Lock the vertex buffer
    hRet = pMesh->LockVertexBuffer( D3DLOCK_READONLY, (void**)&pVertices );
    if ( FAILED(hRet) ) { pMesh->Release(); return hRet; }
```

Now we will iterate through every vertex in the vertex buffer and copy the positional data into a D3DXVECTOR3 called vecPos.

```
    // Compute this frame's bounding box
    for ( i = 0; i < nVertexCount; ++i, pVertices += nVertexStride )
    {
        // Retrieve the position of this vertex in it's reference pose
        vecPos = *(D3DXVECTOR3*)pVertices;
```

We then test the components of this vertex against the currently recorded maximum and minimum box extents and adjust them accordingly.

```
        // Test it against the frame bounding box and update if necessary
        if ( vecPos.x < m_vecBoundsMin.x ) m_vecBoundsMin.x = vecPos.x;
        if ( vecPos.y < m_vecBoundsMin.y ) m_vecBoundsMin.y = vecPos.y;
        if ( vecPos.z < m_vecBoundsMin.z ) m_vecBoundsMin.z = vecPos.z;
        if ( vecPos.x > m_vecBoundsMax.x ) m_vecBoundsMax.x = vecPos.x;
        if ( vecPos.y > m_vecBoundsMax.y ) m_vecBoundsMax.y = vecPos.y;
        if ( vecPos.z > m_vecBoundsMax.z ) m_vecBoundsMax.z = vecPos.z;

    } // Next Vertex
```

After this loop finishes, the bounding box will fit every vertex in the mesh and our job is complete. All we have to do now is unlock the vertex buffer and return.

```
    // Unlock the vertex buffer
    pMesh->UnlockVertexBuffer();

    // Release the pointer to the mesh we retrieved
```

```
    pMesh->Release();

    // Success!!
    return D3D_OK;
}
```

It should be noted that while this method need never be called if you are populating the CTriMesh using its CTriMesh::LoadMeshFromX method (it calls this method automatically), if you are procedurally building a CTriMesh then you will want to call this method after you have added all the vertex data and built its underlying D3DXMesh. Another time you will want to call this method is if you detach the underlying D3DXMesh and attach a new one.

GetBoundingBox - CTriMesh

Although the CTriMesh object stores its model space bounding box, we want a function that can optionally return the world space bounding box if needed. A CTriMesh object has no information about where the mesh lives in the world, so we will add a matrix pointer parameter to its GetBoundingBox function. This allows the application to pass a world matrix that will be used to transform the model space bounding box into world space before returning it to the caller. Luckily, we have already written the TransformAABB method that performs this very task and as such, the CTriMesh::GetBoundingBox uses it to optionally perform the world space transformation.

```
void CTriMesh::GetBoundingBox( D3DXVECTOR3 & BoundsMin,
                               D3DXVECTOR3 & BoundsMax,
                               D3DXMATRIX * pMatrix /* = NULL */ ) const
{
    BoundsMin = m_vecBoundsMin;
    BoundsMax = m_vecBoundsMax;

    // Transform if a matrix is provided
    if ( pMatrix ) MathUtility::TransformAABB( BoundsMin, BoundsMax, *pMatrix );
}
```

As you can see, the function is passed the 3D vectors that will receive the resulting AABB extents. First we copy the model space bounding box of the mesh into these vectors. If the caller passed a matrix, this bounding box is then transformed by that matrix prior to the function returning.

15.8 CActor Revisited – World Space Bounding Boxes

Adding a function to our actor that will return its world space bounding box will not be quite as simple as it was for our mesh. The bounding box it returns must be large enough to encompass all the meshes contained in its hierarchy. At first we might think that this could be done at actor creation time by traversing the tree to find all mesh containers and then building a box to contain the vertices in those meshes. That would certainly work if the actor was not animated, but as we know, actors can be

animated and this changes things. If we imagine an actor that has been created to store a hierarchy of meshes that all rotate and move, we can see that the size of the actor's bounding box would changed during animation updates. Imagine a skinned character for example. It would have one bounding box when its arms are by its side and a different one when it holds its arms out to each side. An even better example of the drastic size changes that can occur to an actor's bounding box would be an actor that models a space ship on a launching pad (like our lab project in Chapter 9). At the start of the animation, the craft would be on the launch pad with a small bounding box encompassing the launch pad and the spaceship. As the animation plays, the spaceship takes off and flies off into the distance. As both the launch pad and the actor are mesh containers within the same actor in this example, the bounding box of the actor would have to dynamically grow to contain the craft as it flies off into the distance.

Traversing the actor hierarchy whenever its animation is updated and calculating a new world space bounding box at the per-mesh level is out of the question if we want to do this quickly. Instead, we will use an approach that will take advantage of the fact that the actor has a spatial tree of its own that can be traversed and updated. Our design approach will be very familiar to you since it is identical in concept to the way we generate the world matrices (combined matrices) for each frame.

Each frame in the tree will now store two sets of bounding box extents (four vectors). Our new D3DXFRAME_MATRIX structure is shown below.

```
struct D3DXFRAME_MATRIX : public D3DXFRAME
{
    D3DXMATRIX   mtxCombined;    // Combined matrix for this frame.
    D3DXVECTOR3  vecBoundsMin;   // Bounding box (in world space)
    D3DXVECTOR3  vecBoundsMax;   // Bounding box (in world space)

    D3DXVECTOR3  vecObjectMin;   // Bounding box (in object space)
    D3DXVECTOR3  vecObjectMax;   // Bounding box (in object space)
    bool         bObjectBounds;  // This bounding box describes the extents
                                 // of a physical object?
};
```

15.8.1 The Model Space Bounding Boxes

vecObjectMin and vecObjectMax will be calculated automatically when the actor's data is first loaded from the X file (via a call from the LoadActorFromX method to the BuildBoundingBoxes method). It will store the model space bounding box of any frame that directly stores mesh data. But what does the bounding box of a frame represent? In our case it will represent a bounding box that is large enough to contain all the mesh containers ***directly*** attached to a given frame. Only frames in the hierarchy that have mesh containers attached will store a model space bounding box, so there will not yet be any parent/child relationship as we see in a spatial tree of bounding volumes.

During the building phase of these boxes, the tree will be traversed and any frame that has a mesh container attached will have a bounding box generated for that mesh (or list of meshes if multiple mesh containers are assigned to the same frame). That model space box with be stored in the owner frame. You will notice that our updated frame structure also stores a new Boolean called bObjectBounds. This

will be set to true only for frames that store a model space bounding box in the vecObjectMin and vecObjectMax members (i.e., frames that have mesh data directly attached to them). However, we must also remember that the actor may contain a skinned mesh, in which case things are a little different.

Up until now we have been talking about the actor and its model space bounding boxes in the simplest form. That is, any frame that has a mesh container attached will also store a model space bounding box describing the position and size of that mesh in the actor's reference pose. However, if we think about the bones of an actor, they do not have mesh containers attached to them (usually the entire skin is stored in the tree attached to an arbitrary frame; most often the root) so does that mean bones should not store model space bounding boxes either? It certainly does not!

Although a bone may not have a mesh container structure attached to it, it does represent some portion of a mesh. For example, imagine a bone that has all the vertices of a skinned character's elbow mapped to it. That bone should contain a bounding box that describes the position and size of the elbow in the default pose. Therefore, we have two cases where a frame will store a model space bounding box. The first is when the frame has a mesh container attached and the second is when the frame is being used as a bone by a skinned mesh. In the latter case, its model space bounding box will bound the section of vertices (in their model space reference pose) in the skin that it influences.

For a skinned character, we can easily imagine having bounding boxes surrounding all the bones in the reference pose (see Figure 15.8). The black box shows what we are trying to ultimately achieve -- a bounding box that encompasses the entire actor, in world space.

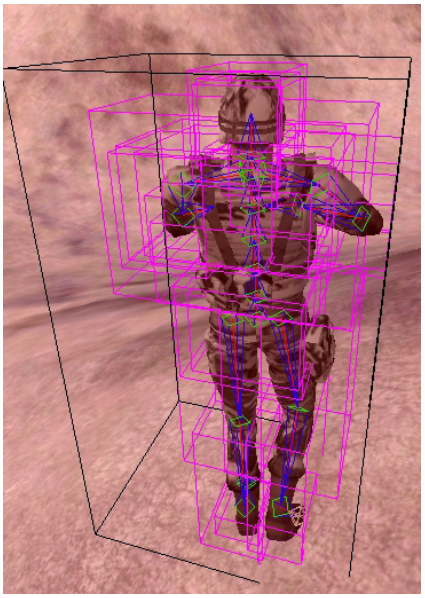


Figure 15.8

So we have determined that the object space bounding boxes of any frame that contains (or is attached to) vertex data will be calculated in a single function called once when the actor is first created. This function is called CActor::BuildBoundingBoxes and it is called from the bottom of the CActor::LoadActorFromX function.

The calculation of these boxes will not be fast enough for real-time work, which is why we are lucky it only has to be performed once. We will have to traverse the tree searching for frames that contain mesh containers and need to have an object space box created for them. When a mesh container is found, we then test to see if the mesh is a skin. If it is not a skin, our task is simple: we lock the mesh's vertex buffer and iterate through each vertex, adjusting the size of the frame's object space box extents until it is large enough to encompass all vertices of that mesh. If multiple mesh containers are attached to the frame, they must all be tested in this way and the bounding box of the frame will grow to fit all these attached meshes.

When the mesh container stores a skin, things are a little bit different. We have to loop through each bone and retrieve the vertices that are influenced by it. Once we have a list of vertices for the current bone we are processing we must transform those vertices into bone space. Remember, the skin itself will be in its own model space at this point and we want its vertices in the space of the actor. After

transforming the vertices attached to the current bone into bone space, we compute its bounding box and grow the extents of the frame's box if it is not large enough to contain it. We do this same step for each bone that influences the mesh.

We will now look at the code to the CActor::BuildBoundingBoxes method. One thing to bear in mind when you see this code is that it never initializes the bounding boxes of any frames to default values. This is done when the frame is first created in the CAllocateHierarchy::CreateFrame callback. This is quite important because a single bone may influence more than one skin. Resetting its bounding box before calculating the extents of any particular skin's vertices would cancel out the contribution of other skins.

BuildBoundingBoxes - CActor

This method is a recursive function that is called from the CActor::LoadActorFromX function. It is passed a pointer to the root frame and then traverses the frame hierarchy looking for frames that contain meshes or that influence the vertices of a skinned mesh. Once found, it grows the model space bounding box of the owner frame to contain any meshes/vertices it directly influences. We will cover the code one section at a time.

The first thing we do is loop through the list of mesh containers stored at the frame. If there are no mesh containers attached to this frame, no action is taken and the frame does not have a model space bounding box calculated for it. At the end of the function, it simply steps into the sibling and child lists.

At the start of the mesh container traversal loop, we determine whether the mesh stored in this container is a progressive mesh or a regular mesh, so that we can access the correct pointer in the D3DXMESHDATA structure. We then query the mesh for a D3DXBASEMESH interface so that we can work with both mesh types through a single interface for the remainder of the function.

```
HRESULT CActor::BuildBoundingBoxes( LPD3DXFRAME pFrame )
{
    HRESULT             hRet;
    D3DXFRAME_MATRIX * pMtxFrame  = (D3DXFRAME_MATRIX*)pFrame;
    D3DXFRAME_MATRIX * pBoneFrame = NULL;
    UCHAR            * pVertices  = NULL;
    ULONG               nVertexStride, nVertexCount, nBoneCount;
    ULONG               nInfluenceCount, i, j;
    D3DXVECTOR3         vecPos;

    // If this has a mesh container , we'll check for skinning etc.
    D3DXMESHCONTAINER_DERIVED * pContainer =
                               (D3DXMESHCONTAINER_DERIVED*)pFrame->pMeshContainer;

    for ( ; pContainer;
            pContainer=(D3DXMESHCONTAINER_DERIVED*)pContainer->pNextMeshContainer )
    {
        // Retrieve the mesh
        LPD3DXBASEMESH pMesh = NULL;
```

```
        switch( pContainer->MeshData.Type )
        {
            case D3DXMESHTYPE_MESH:

                // Skip if no mesh stored here
                if ( !pContainer->MeshData.pMesh ) continue;

                // Query the interface to get back the base mesh.
                hRet =
                pContainer->MeshData.pMesh->QueryInterface( IID_ID3DXBaseMesh,
                                                            (void**)&pMesh );
                if ( FAILED( hRet ) ) return hRet;
                break;

            case D3DXMESHTYPE_PMESH:

                // Skip if no mesh stored here
                if ( !pContainer->MeshData.pPMesh ) continue;

                // Query the interface to get back the progressive mesh
                hRet =
                pContainer->MeshData.pPMesh->QueryInterface( IID_ID3DXBaseMesh,
                                                             (void**)&pMesh );
                if ( FAILED( hRet ) ) return hRet;
                break;

            default:

                // We don't support other types
                continue;

        } // End mesh type
```

At this point we have our mesh pointer, so we will retrieve the vertex count of its vertex buffer and the stride of each vertex before locking the vertex buffer.

```
        // Retrieve additional info we need for processing
        nVertexStride = pMesh->GetNumBytesPerVertex( );
        nVertexCount  = pMesh->GetNumVertices( );

        // Lock the vertex buffer
        hRet = pMesh->LockVertexBuffer( D3DLOCK_READONLY, (void**)&pVertices );
        if ( FAILED(hRet) ) { pMesh->Release(); return hRet; }
```

Now that we have a pointer to the vertices, we have to figure out if this is a normal mesh attached to this frame or if the mesh is a skin that may be influenced by many other bones in the hierarchy. If a skin is not stored here then we know that the container's pSkinInfo pointer will be NULL. In the code snippet we see the code that handles the non-skin case. It loops through every vertex in the vertex buffer and grows the frame's bounding box to encompass the vertices. As mentioned previously, notice that we do not initialize the frame's object space bounding box extents. This is done when the frame itself is first created in the CAllocateHierarchy::CreateFrame method (called by D3DX during the loading of the hierarchy). We must not do that here as this may be one of many meshes attached to this frame. As we

know, it is possible for a frame to have a whole list of mesh containers attached via their pNextMeshContainer pointers.

```
        // Skinned mesh?
        if ( !pContainer->pSkinInfo )
        {
            // Compute this frame's bounding box
            // Note: This frame may have more than one mesh container,
            // boxes initialized during the 'CAllocateHierarchy::CreateFrame' call.
            for ( i = 0; i < nVertexCount; ++i, pVertices += nVertexStride )
            {
                // Retrieve the position of this vertex in it's reference pose
                vecPos = *(D3DXVECTOR3*)pVertices;

                // Test it against the frame bounding box and update if necessary
                if ( vecPos.x < pMtxFrame->vecObjectMin.x )
                     pMtxFrame->vecObjectMin.x = vecPos.x;

                if ( vecPos.y < pMtxFrame->vecObjectMin.y )
                     pMtxFrame->vecObjectMin.y = vecPos.y;

                if ( vecPos.z < pMtxFrame->vecObjectMin.z )
                     pMtxFrame->vecObjectMin.z = vecPos.z;

                if ( vecPos.x > pMtxFrame->vecObjectMax.x )
                     pMtxFrame->vecObjectMax.x = vecPos.x;

                if ( vecPos.y > pMtxFrame->vecObjectMax.y )
                    pMtxFrame->vecObjectMax.y = vecPos.y;

                if ( vecPos.z > pMtxFrame->vecObjectMax.z )
                    pMtxFrame->vecObjectMax.z = vecPos.z;

            } // Next Vertex

            // Frame has applicable bounding box
            pMtxFrame->bObjectBounds = true;

        } // End if standard mesh
```

As you can see in the above code, after every vertex has been tested against the current extents of the frame's bounding box (and adjusted where necessary), we set the frame's bObjectBounds member to true. We will see later that this Boolean is what will tell our CActor::UpdateFrames method that the frame contains an object space bounding box which must be transformed into world space and propagated up the tree to the root node (more on this later).

The next section of code shows what happens when the mesh container stores a skin. In this case, we first use the ID3DXSkinInfo::GetNumBones method to retrieve the number of bones (frames) in the hierarchy that influence the vertices in this mesh. We then initiate a loop to step through each bone.

```
        else
        {
            LPD3DXSKININFO pSkinInfo = pContainer->pSkinInfo;
```

```
            // Loop through each bone in the skin info
            nBoneCount = pSkinInfo->GetNumBones();

            for ( i = 0; i < nBoneCount; ++i )
            {
```

Now that we know the index of the current bone we are processing in the ID3DXSkinInfo's array of bone information, we call its GetBoneName method to retrieve the name of the current frame we are processing. This is then passed into the CActor::GetFrameByName method which traverses the hierarchy searching for the frame and returns a pointer to that frame when found.

```
                // Attempt to retrieve the frame for this bone
                pBoneFrame=(D3DXFRAME_MATRIX*)
                           GetFrameByName(pSkinInfo->GetBoneName( i ) );

                if ( !pBoneFrame ) continue;

                // Retrieve the total number of vertices that this bone influences
                nInfluenceCount = pSkinInfo->GetNumBoneInfluences( i );

                if ( nInfluenceCount == 0 ) continue;
```

After we have a pointer to the current frame/bone we are processing, we pass its index into the ID3DXSkinInfo::GetNumBoneInfluences method. This will return the number of vertices in this skin that are influenced by that bone (which we store in nInfluenceCount). If the return value is zero, then this bone does not influence the skin in any way and we can skip to the next iteration of the loop and process the next bone.

Now that we know how many vertices in the skin are influenced by this bone, we need to fetch the indices of these vertices so that we can adjust the bone's object space bounding box. Therefore, we allocate an array of indices large enough to hold an index for each vertex influenced by the current bone and we allocate an array of floats that will be used to store the weight for each vertex. We pass pointers to these two arrays, along with the bone index, into the ID3DXSkinInfo::GetBoneInfluence method. This function will fill the passed arrays with the indices and weights of all vertices that are influenced by the current bone we are processing.

```
                // Allocate enough space for influences
                ULONG * pIndices = new ULONG[ nInfluenceCount ];
                if ( !pIndices )
                { pMesh->UnlockVertexBuffer();
                  pMesh->Release();
                  return E_OUTOFMEMORY;
                }

                float * pWeights = new float[ nInfluenceCount ];
                if ( !pWeights )
                {
                    delete []pIndices;
                    pMesh->UnlockVertexBuffer();
                    pMesh->Release();
```

```
                    return E_OUTOFMEMORY; }

                // Retrieve the influence array
                hRet = pSkinInfo->GetBoneInfluence( i, pIndices, pWeights );
                if ( FAILED(hRet) )
                {
                    // Clean up and continue
                    delete []pIndices;
                    delete []pWeights;
                    pMesh->UnlockVertexBuffer();
                    pMesh->Release();
                    continue;

                } // End if failed to get influences
```

Now that we have the array of indices, we can loop through each one and fetch the position of the vertex at the corresponding index from the vertex buffer. We will store this in a temporary 3D vector, as shown below.

```
                // Loop through each influence
                for ( j = 0; j < nInfluenceCount; ++j )
                {
                    // Retrieve vertex in it's correct reference pose
                    vecPos = (D3DXVECTOR3&)(pVertices[pIndices[j]*nVertexStride]);
```

We now have the vertex position, but this in the local space of the skin model itself, not in the local space of the frame/bone to which it is attached. Therefore, we retrieve the bone offset matrix for the current bone and transform the vertex into bone space. We then test the position of the vertex against the object space (bone space) bounding box and grow its extents where necessary.

```
                    D3DXVec3TransformCoord( &vecPos,
                                            &vecPos,
                                            &pContainer->pBoneOffset[i] );

                    // Test against the frame bounding box and update if necessary
                    if ( vecPos.x < pBoneFrame->vecObjectMin.x )
                        pBoneFrame->vecObjectMin.x = vecPos.x;

                    if ( vecPos.y < pBoneFrame->vecObjectMin.y )
                        pBoneFrame->vecObjectMin.y = vecPos.y;

                    if ( vecPos.z < pBoneFrame->vecObjectMin.z )
                        pBoneFrame->vecObjectMin.z = vecPos.z;

                    if ( vecPos.x > pBoneFrame->vecObjectMax.x )
                        pBoneFrame->vecObjectMax.x = vecPos.x;

                    if ( vecPos.y > pBoneFrame->vecObjectMax.y )
                        pBoneFrame->vecObjectMax.y = vecPos.y;

                    if ( vecPos.z > pBoneFrame->vecObjectMax.z )
                        pBoneFrame->vecObjectMax.z = vecPos.z;
```

```
                } // Next Influence
```

After the index loop ends, we have correctly adjusted the object space bounding box for the current bone to contain all the vertices in the skin that are influenced by it. All we have to do now is set the bone's pObjectBounds Boolean to true and delete the temporary index and weight arrays.

```
                // Frame has applicable bounding box
                pBoneFrame->bObjectBounds = true;

                // Clean up memory
                delete []pIndices;
                delete []pWeights;

            } // Next Bone

        } // End if skinned mesh

        // Unlock the vertex buffer
        pMesh->UnlockVertexBuffer();

        // Release the pointer to the mesh we retrieved
        pMesh->Release();

    } // Next mesh container
```

After we have processed every bone that influences the skin and calculated the bounding box for each one, we unlock the vertex buffer and release the mesh interface we acquired.

Finally, the function continues its traversal of the tree looking for more mesh containers.

```
    // Has a sibling frame?
    if (pFrame->pFrameSibling != NULL)
    {
        hRet = BuildBoundingBoxes( pFrame->pFrameSibling );
        if ( FAILED(hRet) ) return hRet;

    } // End if has sibling

    // Has a child frame?
    if (pFrame->pFrameFirstChild != NULL)
    {
        hRet = BuildBoundingBoxes( pFrame->pFrameFirstChild );
        if ( FAILED(hRet) ) return hRet;

    } // End if has child

    // Success!!
    return D3D_OK;
}
```

Any frames which do not have meshes attached to them or are not used to influence skins do not have an object space bounding box. Consequently, they will have their bObjectBounds Booleans set to false.

15.8.2 World Space Bounding Boxes

So far we have an object space bounding box stored at every frame that contains a mesh. The problem is that we need our actor to return a world space bounding box for the entire actor that will automatically have its size updated when the actor animates. We can see in Figure 15.9 that when the character is animated, the actor's bounding box changes in size to encompass the world space bounding boxes of all its child frames.

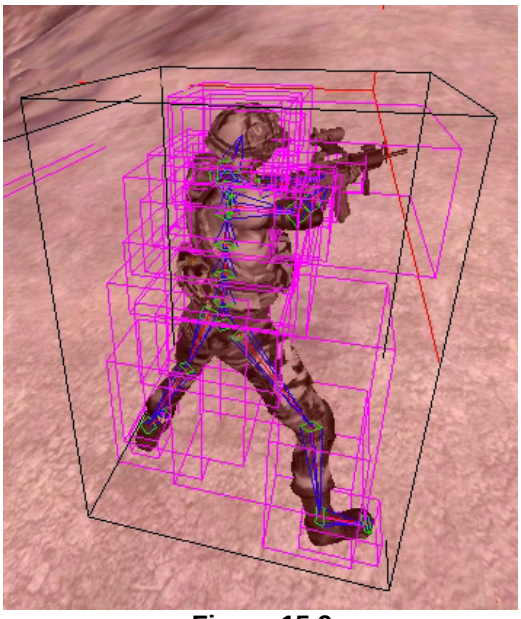


Figure 15.9

In Figure 15.9, the purple bounding boxes are *not* the bounding boxes we have just calculated; they are those bounding boxes after they have been transformed into world space. These world space bounding boxes (which need to be updated every time the actor is animated) are then merged together to compute the final world space bounding box for the entire actor.

The vecBoundsMax and vecBoundsMin members of our frame structure will be updated every time the actor is updated inside the CActor::UpdateFrameMatrices call. This is the method that traverses the hierarchy, combining matrices to ultimately store a world matrix at each frame. In many ways, our bounding boxes will work in exactly the same way as the relative and absolute matrices we are used to working with.

Recall that when the actor is first created, each frame contains a matrix that describes its position in parent relative space. However, each frame also stores another matrix that is updated every time the actor is moved. This update happens in the UpdateFrameMatrices method. You will recall that this function is passed a world matrix which is initially combined with the root frame's matrix to generate the world matrix of the root frame. This world matrix is then passed down to the children, where it is combined with their relative matrices to create the world matrices for each frame, and so on right down the tree. When UpdateFrameMatrices returns, every frame in the tree will store an absolute world matrix.

Our bounding boxes will basically work the same way. Our actor will store bounding box extents which will be used to store the bounding box of the entire actor. These will be updated every time the application calls the SetWorldMatrix method, which as we know, calls the UpdateFrameMatrices method. The UpdateFrameMatrices method will now have some code added so that as it steps through the tree, it will calculate the world space bounding box for every frame in the tree. The world space bounding box of a frame will be a box that has been adjusted to contain the world space bounding boxes of all frames beneath it in the hierarchy. When this process works its way back up the root, we will have a box that is in world space that encompasses every frame in the tree. The following description takes you through how it will work…

As we step into a frame we will calculate its absolute world matrix and store it in the frame as normal. We will then traverse into the sibling and child lists. When a frame is reached that has its bObjectBounds Boolean set to true, we will transform its object space bounding box into world space using our TransformAABB method. We will then store the world space bounding box in the frame and will pass it back to the parent frame when the function returns. The parent frame, having received the world space bounding box of its child(ren) will then adjust the returned box by transforming its own object space bounding box into world space (if it contains one) and grow the box to contain both its own box and the box returned by its child. The combined world space box will then be returned to its parent where the same happens again.

By positioning the bounding box propagation code after the lines that traverse into the child and sibling lists, we make sure that we are calculating the bounding box *not* on the way down the tree (as is the case with the frame matrices), but on our way back *up* the tree. As we begin unwinding the call stack and work our way back up the tree, we can combine the world space bounding box returned by the children with the current frame's own world space bounding box. Of course the world space bounding box returned by the children will also have been combined with the world space bounding boxes of all their children, and so on. What we end up with is a spatial hierarchy; a tree of bounding boxes such that every frame in the tree stores a world space bounding box that encompasses the world space bounding boxes of all the frames beneath it in the tree. The bounding box finally returned from the CActor::UpdateFrameMatrices method back to the CActor::SetWorldMatrix method is the world space bounding box of the entire actor in its current pose. We can then store this information away in the actor's bounding box extents member variables.

> **Note:** This bottom-up calculation approach is commonly used when assembling other types of popular bounding volume hierarchies like scene graphs, AABB trees, etc. We will examine some of these other hierarchy types in Module III of this series.

CActor has had two new members added to store the world space bounding box of the actor in its current pose. These two vectors will be populated by the data returned from the UpdateFrameMatrices call.

Excerpt From CActor

```
D3DXVECTOR3  m_vecBoundsMin;   // Minimum bounding box extents (in world space)
D3DXVECTOR3  m_vecBoundsMax;   // Maximum bounding box extents (in world space)

void   GetBoundingBox ( D3DXVECTOR3 & BoundsMin, D3DXVECTOR3 & BoundsMax ) const;
void   UpdateFrameMatrices(        LPD3DXFRAME   pFrame,
                           const   D3DXMATRIX  * pParentMatrix,
                                   D3DXVECTOR3 * pFrameMin /* = NULL */,
                                   D3DXVECTOR3 * pFrameMax /* = NULL */ )
```

CActor has also had two new method added, which we will discuss next.

GetBoundingBox - CActor

The GetBoundingBox method returns the world space bounding box for the actor. We saw earlier that our CScene::AnimateObjects method called this function to send the box into the ISpatialTree:UpdateTreeObject method so that the leaf list for the object could be updated.

```
void CActor::GetBoundingBox( D3DXVECTOR3 &BoundsMin, D3DXVECTOR3 &BoundsMax ) const
{
    BoundsMin = m_vecBoundsMin;
    BoundsMax = m_vecBoundsMax;
}
```

This method will only be valid once the application has called CActor::SetWorldMatrix or one of the other CActor methods that automatically calls the UpdateFrameMatrices method. It will always contain the world space bounding box calculated during the last CActor::SetWorldMatrix call.

SetWorldMatrix – CActor (Updated)

The application uses the CActor::SetWorldMatrix method to set the world matrix of the actor and optionally rebuild the world matrices of each frame in the hierarchy. This new version of the code initializes the actor's bounding box extents and then sends them into the UpdateFrameMatrices method.

```
void CActor::SetWorldMatrix( const D3DXMATRIX * mtxWorld /* = NULL */,
                             bool UpdateFrames /* = false */ )
{
    // Store the currently set world matrix
    if ( mtxWorld )
        m_mtxWorld = *mtxWorld;
    else
        D3DXMatrixIdentity( &m_mtxWorld );

    // Update the frame matrices
    if ( IsLoaded() && UpdateFrames )
    {
        // Reset the bounding box
        m_vecBoundsMin = D3DXVECTOR3(  FLT_MAX,  FLT_MAX,  FLT_MAX );
        m_vecBoundsMax = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX );

        UpdateFrameMatrices( m_pFrameRoot,
                             mtxWorld,
                             &m_vecBoundsMin,
                             &m_vecBoundsMax );

    } // End if update frames

}
```

As you can see, the CActor::UpdateFrameMatrices method now supports two additional parameters that allow you to pass in two extent vectors to be filled with the world space bounding box of the passed frame. As the frame we are passing is the root node, and the vectors we are passing are the actor's world space extent vectors, we know that on function return the CActor's m_vecBoundsMin and m_vecBoundsMax vectors will contain the world space bounding box of the root node. This is the world space bounding box that encompasses the entire hierarchy of the actor and all its contained meshes.

UpdateFrameMatrix - CActor

This is the function where everything comes together. On top of its original job of calculating the world matrices at each frame, it now has to transform the object bounding box of any frame that contains mesh data into world space and return it back to its parent. Because we want the boxes to be calculated after we have returned from traversing the child list, the code for transforming the box will be at the bottom of the function.

The function is now passed pointers two extent vectors which will store the world space bounding box for the current frame being visited in any given recursion of the function.

```
void CActor::UpdateFrameMatrices( LPD3DXFRAME pFrame,
                                  const D3DXMATRIX * pParentMatrix,
                                        D3DXVECTOR3 * pFrameMin /* = NULL */,
                                        D3DXVECTOR3 * pFrameMax /* = NULL */ )
{
    D3DXFRAME_MATRIX * pMtxFrame = (D3DXFRAME_MATRIX*)pFrame;

    D3DXVECTOR3 vecBoundsMin = D3DXVECTOR3(  FLT_MAX,  FLT_MAX,  FLT_MAX );
    D3DXVECTOR3 vecBoundsMax = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX );;

    if ( pParentMatrix != NULL)
        D3DXMatrixMultiply( &pMtxFrame->mtxCombined,
                            &pMtxFrame->TransformationMatrix, pParentMatrix);
    else
        pMtxFrame->mtxCombined = pMtxFrame->TransformationMatrix;

    // Move onto sibling frame
    if ( pMtxFrame->pFrameSibling ) UpdateFrameMatrices( pMtxFrame->pFrameSibling,
                                                         pParentMatrix,
                                                         pFrameMin,
                                                         pFrameMax );

    // Move onto first child frame
    if ( pMtxFrame->pFrameFirstChild )
         UpdateFrameMatrices( pMtxFrame->pFrameFirstChild,
                              &pMtxFrame->mtxCombined,
                              &vecBoundsMin,
                              &vecBoundsMax );
```

The first part of the function has hardly changed, with the exception that we now initialize temporary bounding box extents vectors and send them into the child recursions. The sibling recursions all share the same parent, so they are passed the same bounding box vectors that were passed into this function.

When the child list recursions return, the local bounding box extent vectors vecBoundsMin and vecBoundsMax will contain a world space bounding box that encompasses all of the children beneath the current frame. To see why this is the case, we have to look at the new second section of the function that calculates a world space bounding box for any mesh data that might be attached to this frame.

If the current frame's bObjectsBounds member is set to true then it means this frame either has a mesh attached or it influences a skin. In either situation, it means we have a frame that contains a model space bounding box. When this is the case, we transform the model space AABB using the world matrix of the current frame we are visiting.

```
    // Should we also update our actor's bounding box with the new information
    if ( pMtxFrame->bObjectBounds )
    {
        D3DXVECTOR3 vecMin = pMtxFrame->vecObjectMin,
                    vecMax = pMtxFrame->vecObjectMax;

        // Transform the mesh's object bounding box into world space
        MathUtility::TransformAABB( vecMin, vecMax, pMtxFrame->mtxCombined );
```

At this point we now have two world space bounding boxes. The first is the world space bounding box of the mesh data stored at this node (which we have just stored in the frame's world space extent vectors). The second, stored in vecBoundsMin and vecBoundsMax, was returned from the child list and bounds all the children of this frame. We will now merge these two boxes together by adjusting the world space bounding box at this frame so that it also encompasses the bounding box returned from the children.

```
        // Test it against the actor bounding box and update if necessary
        if ( vecMin.x < vecBoundsMin.x ) vecBoundsMin.x = vecMin.x;
        if ( vecMin.y < vecBoundsMin.y ) vecBoundsMin.y = vecMin.y;
        if ( vecMin.z < vecBoundsMin.z ) vecBoundsMin.z = vecMin.z;
        if ( vecMax.x > vecBoundsMax.x ) vecBoundsMax.x = vecMax.x;
        if ( vecMax.y > vecBoundsMax.y ) vecBoundsMax.y = vecMax.y;
        if ( vecMax.z > vecBoundsMax.z ) vecBoundsMax.z = vecMax.z;

    } // End if apply frame's bounding box
```

The frame now contains the world space bounding box for its mesh data stored and all of the meshes stored below it in the tree. Notice in the above conditional code that what we have actually done is grown the temporary bounding box that was returned from the child list to contain the bounding box we calculated for this frame's mesh data. The next line of code describes why we adjusted the local temporary box and not the box stored in the frame.

```
    // Store the bounding box in this frame
    pMtxFrame->vecBoundsMin = vecBoundsMin;
    pMtxFrame->vecBoundsMax = vecBoundsMax;
```

As you can see, the next step is to copy the temporary bounding box which we just calculated into the frame's world space box extents. However, this code is not in a conditional code block and will be executed even if the frame has no mesh data attached. If the frame has no mesh data, then it is simply assigned the world space box that was returned from the child list. This is actually true in both cases, but in the case where we have mesh data assigned, we simply adjust it first by the mesh's world space bounds before assigning it.

This function is also expected to return the bounding box of this frame back to the parent so that the parent can combine the box with its mesh data, and so on back up to the root. As you can see, that last step is to copy this frame's world space bounding box into the passed vector extents so that the parent will have them on function return.

```
    // Test it against the parent's bounding box and update if necessary
    if ( pFrameMin )
    {
        if ( vecBoundsMin.x < pFrameMin->x ) pFrameMin->x = vecBoundsMin.x;
        if ( vecBoundsMin.y < pFrameMin->y ) pFrameMin->y = vecBoundsMin.y;
        if ( vecBoundsMin.z < pFrameMin->z ) pFrameMin->z = vecBoundsMin.z;

    } // End if frame min extents requested

    if ( pFrameMax )
    {
        if ( vecBoundsMax.x > pFrameMax->x ) pFrameMax->x = vecBoundsMax.x;
        if ( vecBoundsMax.y > pFrameMax->y ) pFrameMax->y = vecBoundsMax.y;
        if ( vecBoundsMax.z > pFrameMax->z ) pFrameMax->z = vecBoundsMax.z;

    } // End if frame max extents requested
```

The final bit of this function is not new; it loops through each mesh container stored at the frame and invalidates it. This is important if software skinning is being used as it tells the actor that any skin stored here will have to be re-transformed into world space before being rendered again.

```
    // If this has a mesh container (or set of containers), we must invalidate them
    D3DXMESHCONTAINER_DERIVED * pContainer =
                            (D3DXMESHCONTAINER_DERIVED*)pMtxFrame->pMeshContainer;

    for (;pContainer;
          pContainer = (D3DXMESHCONTAINER_DERIVED*)pContainer->pNextMeshContainer )
    {
        // Flag as invalid
        pContainer->Invalidated = true;

    } // Next Container
}
```

We have now covered all the new actor code that calculates and returns the world space bounding box needed in order to use the actor as a dynamic object in our spatial tree. Of course, by implementing the hierarchy of bounding boxes the way we have, it means that we now have the ability to hierarchically frustum cull an actor even without using our spatial tree system.

For example, imagine you have loaded a single X file containing the entire scene. This scene, with hundreds of meshes and animations would now be contained in a single actor. Every time we update the actor, a world space bounding box is stored at each frame. This box contains the bounding boxes of all frames underneath it (just like a quad-tree or an oct-tree), so you could easily write a function that will traverse the frames of the hierarchy with a camera and perform hierarchical frustum rejection using the techniques we have covered in this chapter (including the optimizations). At each frame, the recursive function could test the world space bounding box of the frame to see if it is within the frustum, if not, the child list does not have to be stepped into possible rejecting hundreds of meshes stored below it. The actor has now become a bounding volume hierarchy (a spatial tree of sorts). You could even run hierarchical bounding volume or ray intersection queries on the actor's meshes if needed. This might be helpful if you wanted to include some form of body part specific damage for example.

15.9 Adding Visibility Determination to CTerrain

A few changes have had to be made to the CTerrain class so that it now makes use of the spatial tree. As we know, each CTerrain is made up of a series of CTerrainBlocks where each block is a rectangular mesh that represents a portion of the terrain. When the terrain blocks are created (in the CTerrain::BuildTerrainBlocks method), each terrain block has its world space bounding box registered with the spatial tree as a detail object. This forces the tree to compile a tree that is large enough to encompass the terrain, even though the terrain polygons are never passed into the tree and compiled into the leaves. As discussed in the previous chapter, the last thing we want is for the spatial tree to be only as large as the static data contained within it. It makes no sense for us to compile the terrain triangles into the tree since each block can be rendered more efficiently as a separate mesh. However, we do want the spatial tree to be large enough to encompass the terrain so that a dynamic object that is placed on that terrain is never in a position where it is not in a region of space that does not belong to the tree. This would mean the dynamic object would not exist in any leaf nodes and as such, could not benefit from the visibility system.

Each terrain block will be given a visibility Boolean which will be registered with the spatial tree as a dynamic/tree object. Although the terrain blocks are not literally dynamic (they will never move), by registering a tree object for each terrain block, we can have the terrain block Boolean pointers stored in leaves and set to true by the tree whenever that terrain block is visible. The CTerrain::Render method can then loop through each terrain block and skip rendering any that have not had their Boolean set to true by the spatial tree.

Below we see the CTerrainBlock member variables and have highlighted the two new ones added in Lab Project 14.1.

Excerpt from CTerrain.h

```
class CTerrainBlock
{
 public:

    ……Methods Not Shown……
```

```
    // Public Variables for This Class
    ULONG                   m_nVertexCount;     // Number of vertices stored
    UCHAR                  *m_pVertex;          // Simple temporary vertex array
    ULONG                   m_nIndexCount;      // Number of indices stored
    USHORT                 *m_pIndex;           // Simple temporary index array
    LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer;    // Vertex Buffer to be Rendered
    LPDIRECT3DINDEXBUFFER9  m_pIndexBuffer;     // Index Buffer to be Rendered
    UCHAR                   m_nStride;          // The stride of each vertex
    ULONG                   m_nFVFCode;         // The flexible vertex format code.

    D3DXVECTOR3             m_BoundsMin;        // Bounding box minimum extents
    D3DXVECTOR3             m_BoundsMax;        // Bounding box maximum extents

    long                    m_nTreeObjectIndex;     // SpatialID
    bool                    m_bVisible;             // Is this object visible?

};
```

long m_nTreeObjectIndex
This member will contain the spatial ID returned from the ISpatialTree::InsertTreeObject method. This method will be called to register the terrain block with the spatial tree as a dynamic/tree object.

bool m_bVisible
This is the Boolean whose address will be stored in the TreeObject when it is registered with the spatial tree. This Boolean will be set to true by the spatial tree during the visibility pass if any leaf in which the terrain block's bounding volume exists is currently visible. The CTerrain::Render function will loop through its array of terrain blocks and only render those which have m_bVisible set to true.

The CTerrain object will also store an ISpatialTree pointer that can be set via a new method called CTerrain::SetSpatialTree. This method simply copies that passed pointer and stores it in a CTerrain member variable. CTerrain::SetSpatialTree is called in the CScene::ProcessEntities method when it encounters a terrain entity and extracts the data to create a new CTerrain object. After the object has been created, the scene's spatial tree pointer will be passed into CTerrain::SetSpatialTree.

The following snippet of code shows a section of the CScene::ProcessEntities function which is called by CScene::LoadSceneFromIWF to extract and process any entities stored in the IWF file. You will recall from our previous discussions that if the entity currently being processed is a terrain entity, its data is extracted into a TERRAINENTITY structure which is then passed to the CTerrain::LoadTerrain method. This method loads the heightmap, textures, and any additional information stored in the passed TERRAINENTITY structure before calling CTerrain::BuildTerrainBlocks to actually create the vertex data for each terrain block.

Excerpt from CScene::ProcessEntities

```
If ( THIS IS AN ENTITY WE ARE PROCESSING)
{
   …Snip… : Extract terrain data from file into TERRAINENTTY structure (Terrain)

// Allocate a new terrain object
pNewTerrain = new CTerrain;
if ( !pNewTerrain ) break;
```

```
// Setup the terrain
pNewTerrain->SetD3DDevice( m_pD3DDevice, m_bHardwareTnL );
pNewTerrain->SetTextureFormat( m_TextureFormats );
pNewTerrain->SetRenderMode( GetGameApp()->GetSinglePass() );
pNewTerrain->SetWorldMatrix( (D3DXMATRIX&)pFileEntity->ObjectMatrix );
pNewTerrain->SetDataPath( m_strDataPath );

// Set the spatial tree, to allow the terrain to build detail areas
pNewTerrain->SetSpatialTree( m_pSpatialTree );

// Store it
m_pTerrain[ m_nTerrainCount - 1 ] = pNewTerrain;

// Load the terrain
if ( !pNewTerrain->LoadTerrain( &Terrain ) ) return false;

// Add to the collision system
m_Collision.AddTerrain( pNewTerrain );
}
```

We have snipped out the code that extracts the entity data from the IWF file and stores it in the TERRAINENTITY structure, but once this is done, a new CTerrain block is allocated and its device, render mode, and world matrix are set. Notice the new line (highlighted in bold) which sets the spatial tree pointer of the CTerrain object to the same spatial tree being used by the application. After this, the new terrain object is added to the scene's CTerrain array before its LoadTerrain method is called to actually build the terrain data and all of the terrain blocks. Finally, as we saw several lesson ago, the terrain object is registered with the collision system.

BuildTerrainBlocks – CTerrain (Updated)

After the terrain information has been loaded from the IWF file and the spatial tree pointer has been set, the CTerrain::LoadTerrain method is called to build the entire terrain and all terrain blocks. This method will call the CTerrain::BuildTerrainBlocks method to build the vertex data for each terrain block.

The next new section of code we will look at is executed in the BuildTerrainBlocks method. It occurs just after the current terrain block being processed has been built. At this point, the spatial tree has not been compiled so we can register the block bounding box as a detail area so that the region of space taken up by this terrain block will be compiled into the tree. The following section of code is actually embedded in a loop that creates each terrain block.

As you can see, we instantiate a TreeDetailArea structure and set its bounding box equal to that of the terrain block. We could store the terrain block pointer in the context pointer but we do not need to. We are only registering this detail area to shape the spatial tree; we do not need to know where it came from.

```
// Building for spatial tree?
if ( m_pSpatialTree )
{
```

```
    TreeDetailArea Area;
    TreeObject     Object;

    // Store detail area bounding box
    Area.BoundsMax = pBlock->m_BoundsMax;
    Area.BoundsMin = pBlock->m_BoundsMin;
    Area.pContext  = NULL;
```

As the bounding box of the terrain block is in model space we should transform it by the CTerrain's world matrix so that it represents the terrain block in world space. We then add the new detail area to the tree to shape the tree during compilation.

```
    // Transform the bounding box into world (tree) space
    MathUtility::TransformAABB( Area.BoundsMin, Area.BoundsMax, m_mtxWorld );

    // Add to the spatial tree
    m_pSpatialTree->AddDetailArea( Area );
```

In order for a terrain block to only be rendered when visible, we will register it as a tree object that has its Boolean pointer pointing at the CTerrain::m_bVisible member. In a moment, you will see that once the tree is built, we will call the ISpatialTree::UpdateTreeObject method to assign the terrain block's Boolean pointer to the leaves in which the terrain block is contained. We cannot do that just yet, since the spatial tree has not been built at this time. All we can do at the moment is create a new tree object in the spatial tree and assign it to point at our terrain block's Boolean pointer. Notice in the following code that our CTerrainBlock class now has an m_nTreeObjectIndex member that is used to store the terrain block's spatial ID.

```
    // Set new tree object details
    Object.pContext  = NULL;
    Object.pbVisible = &pBlock->m_bVisible;

    // Add as a tree object
    pBlock->m_nTreeObjectIndex = m_pSpatialTree->InsertTreeObject( Object );

} // End if spatial tree set
```

That is the only new section of code in the CTerrain::BuildTerrainBlocks method. At this point the terrain blocks is registered with the system but does not yet have its Boolean assigned to any leaves in the tree (no leaves exist yet since the tree has not been compiled).

After CScene::LoadSceneFromIWF has called all the processing methods to extract the IWF data, all objects contained in the file that needed to be created will have been. This means that any terrain entities that were in the file will have already been used to create terrains and the terrain blocks of each terrain will have been registered as detail areas and as dynamic objects.

It is now time to build the spatial tree as we saw in the previous lesson. However, we have an additional step to take with the scene's CTerrain array. Although the terrain blocks are already registered with the spatial tree, we had to wait until the tree was compiled before we could assign the terrain blocks to their relevant leaves. Thus, in the small section of code shown below (at the bottom of

CScene::LoadSceneFromIWF), you can see that after the spatial tree is compiled, we loop through each terrain in the terrain array and call its TreeBuildComplete method (a new method).

Excerpt from CScene::LoadSceneFromIWF

```
…SNIP : All Process…… Functions Called Here

// Build the spatial tree and notify the collision engine
if ( !m_pSpatialTree->Build( ) ) return false;

// Notify terrain that objects the spatial tree has been built
for ( i = 0; i < m_nTerrainCount; ++i ) m_pTerrain[i]->TreeBuildComplete();
```

CTerrain::TreeBuildComplete allows the terrain to perform processing after the spatial tree has been compiled. Let us take a look at this simple member function.

TreeBuildComplete – CTerrain

For each terrain block, this function exacts its model space bounding box and transforms it into world space using the world matrix of the terrain. It then uses the ISpatialTree::UpdateTreeObject method to assign the terrain block to its relevant leaves in the tree by passing in the spatial ID of the terrain block and its world space bounding box. This is no different from how we update the positions of our dynamic CObjects in the CScene::AnimateObjects method. The reason we do it here is so that it only ever has to be performed once. Our terrains will never move or be animated, so we make this a one-off leaf assignment function that is called post-build.

```
void CTerrain::TreeBuildComplete( )
{
    ULONG i;

    // Validate
    if ( !m_pSpatialTree ) return;

    // Render Each block
    for ( i = 0; i < m_nBlockCount; i++ )
    {
        D3DXVECTOR3 vecMin = m_pTerrainBlocks[i]->m_BoundsMin;
        D3DXVECTOR3 vecMax = m_pTerrainBlocks[i]->m_BoundsMax;

        // Transform the bounding box
        MathUtility::TransformAABB( vecMin, vecMax, m_mtxWorld );

        // Ask tree to update so that the block is inserted into it's final leaves
        m_pSpatialTree->UpdateTreeObject( m_pTerrainBlocks[i]->m_nTreeObjectIndex,
                                          vecMin, vecMax );

    } // Next Block
}
```

Finally, although we will not show the code here due to the trivial changes, the CTerrain::Render method has now been updated to only render terrain blocks that have their Booleans set to true. These Booleans will be set to true for any terrain blocks that are found to be in visible leaves when the application issues a call to ISpatialTree::ProcessVisibility.

Conclusion

In this chapter we have made significant progress with our rendering technologies and our geometry management. We have implemented a rendering system that will perform hierarchical frustum culling and efficiently collect and draw the static geometry in visible leaves. The rendering system in CBaseTree is certainly complex, and to fully understand the code you will probably need several passes through this book and the source code. Fortunately, with this core technology in place, we now have a system that will be carried into the remaining chapters of this course where we will take visibility processing to the next level.

So far, our visibility system has only partly lived up to its name. After all, it does not render only the visible polygons; it actually renders all polygons inside the view frustum, which is not nearly the same thing. If we imagine being in a small room (located within a larger complex) with no windows and maybe a small door, all that would actually be visible to us would be the four walls of the room and maybe a small section of the corridor we can observe from within the room. Using our current system, if we had a frustum with a far plane positioned a long way in the distance, many of the rooms in this example which are not visible would still be rendered since they fall within the frustum and are therefore considered visible in our system implementation. In truth, all we have really done in this chapter is laid the foundation for a proper visibility system. Our final visibility system will be implemented in the next two chapters of this course.

While we have learned how to efficiently render only what is within the camera's FOV, we have not taken into account that some portions of the level will be obscured by others. If our camera is located in front of a huge wall, ideally we should only render the wall. On the other side of that wall there might be a vast terrain or multiple buildings that should never be seen as they are occluded by the wall in front of us. Our current visibility system does not make provisions for this scenario. We have taken only a preliminary step towards determining a proper potential visibility set. That is, we certainly have narrowed things down to a set of polygons that are potentially visible using our frustum tests (and we have definitely rejected quite a lot of polygons that definitely are not visible), but many of those potentially visible polygons can also be removed from consideration with just a bit more work on our part.

The remainder of this course will work towards the goal of building a PVS (Potential Visibility Set) calculator that will pre-calculate at compile time exactly which leaves could possibly be visible from any other leaf in the scene by factoring in the notion of occlusion. PVS systems have been the backbone of commercial computer games since the days of Quake™. They are what allow seemingly massive environments comprised of a very high number of polygons to be rendered at real-time interactive frame rates. With a PVS system in place, the speed at which your game runs is no longer bound purely by the number of polygons in your scene, or even the number of polygons contained in the frustum. Rather, we

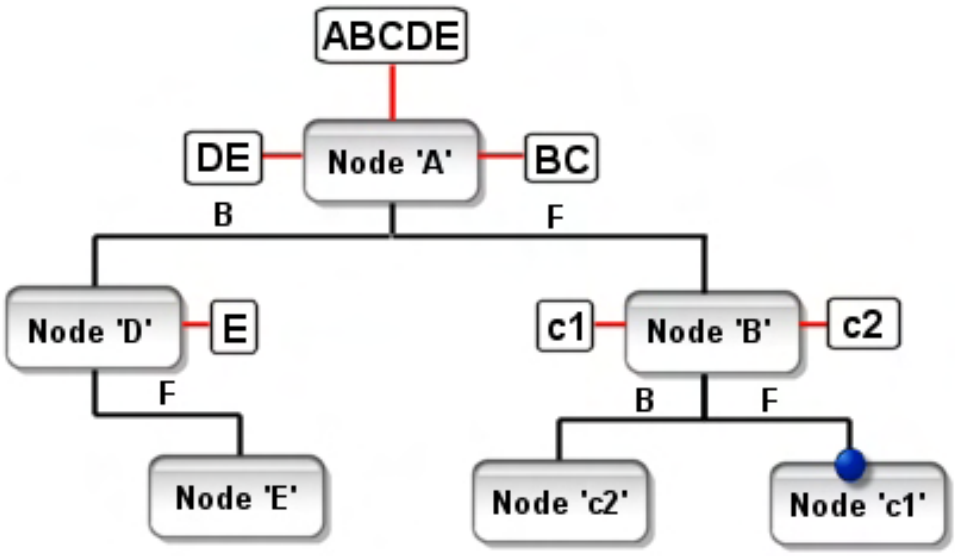
are dealing with only the polygons that can be physically seen by the viewer, which is obviously going to be a far smaller number in the general case. If you consider a room with no windows and closed doors, regardless of the size of the level and the number of polygons it contains, our PVS-based system would only have to render the four walls, the floor and the ceiling which the camera can currently see (the hardware clipping can handle any final bits of actual visibility testing within the room). So the next two lessons will open up a wide array of possibilities in terms of the art assets your engine will be able to handle. You will soon be able to draw incredibly high polygon count scenes that would otherwise be unthinkable, even on today's cutting edge hardware.

In the next chapter we will introduce BSP trees as they are going to be the foundation tree for the PVS system we will implement. We will also learn many other techniques that the BSP tree can be used for, such as perfectly sorting alpha polygons and rendering them in the correct order (something that our hash table design from Module I could not guarantee in all cases). We will also discover that BSP trees can be used to carve meshes from one another or fuse multiple meshes into a single mesh. BSP trees are prevalent in computer science and particularly in game development and they will be one of the most useful tools you will learn about in your training as a game developer.

Make sure that you fully understand how our rendering system works before moving on to the next chapter. We will be using this same system again as we move forward with BSP trees (at least initially), so it is important that you devote the necessary time to working through the system. While this was undeniably a fairly complex implementation (especially once the multiple vertex buffer support was introduced), the underlying logic is still quite straightforward. If you take your time and make sure that you are comfortable with the concepts we have covered here, you will be in good shape in the discussions to come.

Chapter Sixteen

Spatial Partitioning III



Introduction

In this lesson we will continue our discussion of spatial partitioning by introducing Binary Space Partitioning trees. Binary Space Partitioning (BSP) is a technique that has been the cornerstone design element in some of the most powerful and successful 3D game engines on the market. When John Carmack and id software used BSP trees for Doom™ and later Quake™, BSP quickly became an industry buzzword. While these games have aged a bit over the years, you may be surprised to learn that virtually all of the latest first person shooters still continue to use BSP based engines in one form or another. With the more recent introduction of pixel shader programs that often perform many complex rendering passes per polygon, eliminating overdraw and reducing the number of polygons that need to pass through these shader programs has once again become the key to fast game performance -- an area in which the BSP trees (along with accompanying technologies) excel.

The reason this spatial tree was not covered in the previous lessons is because it really does deserve its own chapter due to the myriad ways in which it can be used and its importance in the field of computer science and game development. Having BSP trees at our disposal will allow us to perform perfect alpha sorting very efficiently, allow us to create spatial trees that have arbitrarily shaped leaves (leaves that are not always just boxes), and will allow us to divide the world into areas that describe whether that space is empty or solid. Empty space is space that is not currently occupied by any geometry and can be freely moved around in by the game characters, where as solid space is space that currently describes an area that contains a solid object (e.g., a wall). Because the BSP tree can be used to discriminate between solid and empty areas in a game world, it can be used to calculate a potential visibility set, which is the goal we will be working towards both in this lesson and the next.

A potential visibility set (PVS) is a pre-compiled set of data built by an application called a PVS Calculator that tells our application which leaves can be seen from every leaf in the tree. If leaf A is not visible from leaf B then leaf A's polygon data will not need to be rendered when the camera is in leaf B. Although this process seems similar to the visibility system we developed in the previous lesson, there is a significant difference -- a PVS calculator accounts for the occlusion of geometry when determining what is visible from a particular leaf. If your scene uses a large number of polygons, but the camera is currently located in a small room with no windows or doors, the only polygons that the visibility system would flag for rendering would be the walls, the floor, and ceiling of the room in which the camera is currently located. This would be true regardless of how many polygons were currently intersecting the view frustum. The polygons situated behind those walls are occluded by the walls and therefore would not be seen (and thus not rendered).

The BSP tree will play a crucial part in the construction of the PVS calculator we will create in the following lesson. It is this usage of the BSP tree and the PVS data that it aids in compiling that allowed games such as Quake™, Quake II™, Half Life™, and Medal of Honor™ (and the dozens of games built around those engines) to run at high frame rates despite the vastness of the levels. It is not overstating the point to say that the ability to calculate and process a potential visibility set is one of the most fundamentally important technologies in a 3D graphics engine; it represents a way to process and render only what is visible with virtually zero run-time processing, thus decoupling the performance of your game from the scene polygon count in the general case.

Later in this lesson we will study another BSP use case commonly seen in many 3D world editing packages, such as GILES™. The ability to carve one solid object from another or to fuse two objects into a single mesh are likely familiar to you if you have spent any significant time working with GILES™ or any other world editor package such as WorldCraft™. Indeed you can create very complex scenes by carving and merging simple mesh objects into more complex shapes. This technique is referred to as Constructive Solid Geometry (or sometimes, Geometric Boolean Operations). These techniques are once again made possible by the solid/empty space discrimination that can be made when the scene data is compiled into one or more BSP trees. In GILES™, each brush is compiled internally as a mini-BSP tree that describes which regions of the brush are in solid and empty space. When two brushes are merged into one (called a *union* operation), the polygons from each brush are sent down the other brush's BSP tree. Any polygons that end up in solid space (in either tree) are deleted, leaving two sets of remaining polygons (the non-deleted polygons from each brush). These can then be collected and added to a new single brush that is the union of the original two. BSP tree based Constructive Solid Geometry (CSG) techniques can be used to carve explosion damage into surrounding geometry, to merge a multi-brush level into a single static mesh, or to remove polygons that are not visible because they are embedded in other objects (e.g., removing the bottom face of a crate that is resting on the floor and would therefore never be seen). The latter process can help reduce the polygon count of our scenes and as you will see later, can be used to mold geometric data into a form that will describe solid and empty areas when compiled into a BSP tree.

At the end of this lesson we will have understood and implemented the following:

- **BSP Node Trees:** This tree type will be used for storing alpha polygons and rendering them in a pixel perfect back to front order. Using the BSP tree for this purpose will allow us to get perfect alpha blending results with all geometry configurations (something we have not yet been able to do in our framework). This tree will be constructed in Lab Project 16.1 to manage any alpha polygons that the scene may contain. The application will use this tree (after all non-transparent geometry has been rendered) to render its alpha list in correct back to front order. We will also develop a rendering solution for the BSP node tree that renders the alpha polygons it contains efficiently, trying to maximize batch rendering.

- **BSP Leaf Trees:** This is a tree that compiles in the same way as the node tree but differs in that it collects and stores the polygons at the terminal nodes (i.e., leaves) of the tree. This type of tree will be very similar to the trees we implemented in the previous lesson (with the kD-tree being its closest relative). Our implementation of such a tree can be derived from CBaseTree and can therefore use all of the same rendering functionality. The difference between a BSP leaf tree and a kD-tree is that the leaves are not necessarily box shaped, but can be arbitrarily sized and shaped.

- **Solid BSP Leaf Trees:** This is essentially the exact same tree as the BSP leaf tree with the exception that it expects to have geometry passed into it that meets certain standards. Such a tree (when supplied with the suitable geometry) will compile the geometry into arbitrarily shaped leaf nodes like the standard leaf tree, but this will be done in such a way that the tree will be able to tell us exactly which areas in the scene are considered solid (e.g., inside a wall) and which are considered empty. In the next lesson we will see that it is this property that will allow us to generate portals that are used by the PVS calculator to build the visibility sets for each leaf in the

tree. That is, it is vital that we have a way to compile a spatial tree that will tell us exactly what is solid and what is not. Using this information, our PVS calculator can calculate, from any leaf, exactly which leaves are occluded due to there being a solid obstruction in the way.

- **Constructive Solid Geometry (Boolean Operations):** It is important that we learn how to build a BSP tree that can provide us with the solid/empty information about the scene since we will need this information in our PVS calculator. As mentioned a moment ago, whether we can do so depends on whether the geometry we are being given meets certain standards that the BSP compiler will expect. The CSG techniques we implement in this lesson will allow us to write routines that attempt to mend data that would otherwise be considered unsuitable for the compilation of a solid/empty BSP tree. We shall see that CSG techniques will ultimately be used to generate the data needed to compile a PVS.

16.1 Introducing BSP Trees

Although the kD-tree we discussed in the previous lesson is in fact a tree that performs a binary spatial partition at each node (and could theoretically be referred to as a BSP tree in that sense), the more industry standard view of the term 'BSP tree' is a tree that partitions space at each node (into two halfspaces – thus, binary) using an arbitrarily oriented split plane. Most often, the term is used to describe a ***polygon-aligned*** BSP tree, where the split plane chosen at each node is not an axis-aligned plane (like we use in the kD-tree) but is taken from the polygon dataset. That is, each polygon input into the compilation process has its plane used as a split plane once. Using such a technique, the number of nodes in the tree will be equal to the number of polygons since each one's plane is used to split a node. Figure 16.1 shows how a polygon-aligned BSP tree would divide the space of a scene composed of five polygons (A through E).

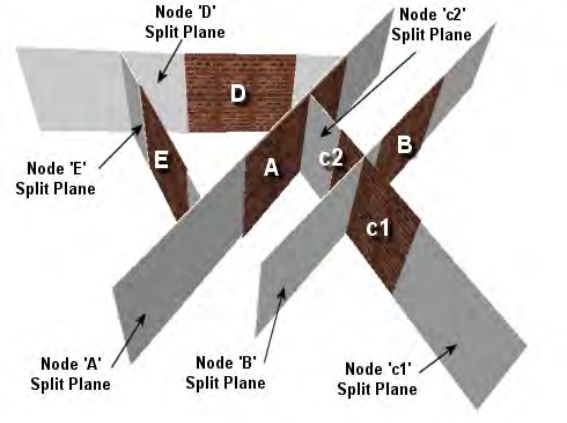


Figure 16.1 : Polygon Aligned BSP Tree

Since the scene is comprised of five polygons (forget about why polygon C is divided into two sub polygons at the moment), five nodes are created. The split plane stored at each node is the plane of the polygon that was selected from the polygon list at that node. As you can see, our scene is divided up into irregularly shaped regions, unlike the quad-tree, oct-tree and kD-tree we discussed in the previous lessons (which always carved the world into axis aligned box shaped regions). However, an important point to remember is that a BSP tree can be compiled using any arbitrarily oriented planes at each node; it need not use the planes of the polygons it has been passed for the compile process. As we will discuss in a moment however, there is great benefit in doing so under certain circumstances.

The compilation of a BSP tree is essentially the same as the building process of any of the other spatial trees we have developed thus far. Each node in the tree stores a split plane that is used to cut the space

represented by that node into two child nodes. While the kD-tree always used a split plane at each node that was aligned with one of the world axes, the BSP tree can use any arbitrarily oriented plane. The polygon list that makes it into that node is then divided into front and back lists, just as we saw with the kD-tree. Any polygons in the list that span the current node's plane are split and the relevant fragments added to both the front and back list. These lists are then passed into the two child nodes where a new plane is selected and the polygon data is further subdivided. This is all identical to the split trees we compiled in the previous lessons, and you will see that the code to both the BSP tree compiler and the kD-tree build function (for example) are almost identical. The only difference in our description so far is that the plane chosen to split each node is not necessarily chosen to be axis aligned; it can be arbitrarily oriented. As one might imagine, using arbitrary split planes at each node no longer divides the underlying region of space into a neatly organized stack of boxes.

There are two main flavors of BSP tree that we will develop in this lesson: node trees and leaf trees. A leaf tree works in the same way as the previous trees we have covered. The terminal nodes of the tree are the leaf nodes and they contain all the polygon data that belongs in that leaf (just as before). These are the polygons that are passed down to that terminal node during the build process. A node tree is similar, but not exactly the same. The only real difference between a leaf tree and a node tree is that the node tree does not contain the leaf structures that store polygon data at the terminal nodes. Instead a node tree might choose to store the polygon data in the nodes themselves or perhaps store no polygon data at all. With this small exception, the node tree is identical to the leaf tree.

In the next section we will introduce the BSP node tree, explain its properties, and see how it can be used for back to front rendering of polygon data. The BSP node tree we implement will generate its node planes from the polygon data compiled into the tree (i.e., a polygon-aligned BSP node tree). The polygons themselves will also be stored at the nodes. That is, each node will contain a split plane that was created by one of the polygons in the original set. That polygon and all others that share that plane will also be stored at the node. Although it might not be obvious why we would want to build a BSP tree that partitions the world using a set of polygon planes or why we would want to store polygons in the nodes at arbitrary levels of the tree (instead of at leaf nodes), all of this will be explained shortly.

16.2 Polygon-Aligned BSP Node Trees

In the early days of 3D game development, when end-users' systems were not equipped with hardware depth buffers and megabytes of video memory, one of the major tasks that faced any game engine programmer was how the scene was going to be rendered into the frame buffer such that polygons nearer the viewer obstructed polygons that were further away. These days, we take for granted the fact that the depth buffer will perform a per-pixel comparison before each pixel is rendered so that a pixel only gets rendered if its camera relative depth is less than the depth of a previous pixel already stored in the frame buffer at that same location. Prior to the introduction of 3D hardware acceleration and the inexpensive main memory found in the mid level PCs of today, 3D engine programmers did not have the luxury of throwing their polygons at the frame buffer in an arbitrary order, relying on the depth buffer to handle the occlusion of distant objects. On early systems, memory was very limited. It was barely possible to fit all the polygon data into memory, let alone allocate another large chunk of memory the size of the frame buffer just to store per-pixel depth information. Additionally, since these engines ran

completely in software, performing a per-pixel test prior to rendering each pixel was also something which could not be performed in real time on early microprocessors.

The 3D developer had bigger concerns than batch rendering by texture or material and in fact, was forced to use a technique that was mutually exclusive with respect to these concepts. The engine had to render the scene so that polygons further away from the viewer were rendered first and polygons nearer the viewer were rendered later. Remember, without a depth buffer, any polygon that was rendered would always be drawn over anything currently in the frame buffer. Even if the polygon that was rendered was further away from the camera position than the polygon currently occupying the same space in the frame buffer, that polygon would be overwritten. In other words, without any depth testing in the pipeline, rendering to the frame buffer was analogous to painting on a canvas. Regardless of the scene already painted on that canvas, anything you painted subsequently would be rendered over the top. Figure 16.2 shows a classic example of what can happen if no depth testing is available and the polygons are rendered in an arbitrary order. In this example we have a mesh of a corridor section consisting of five polygons. The long darker horizontal polygon is the polygon that forms the wall at the end of the corridor. Technically, it should be occluded by the left and right wall sections. However, because it was the last polygon that the artist added to the mesh, and we are simply rendering the faces of the mesh in the order in which they are stored, it is rendered to the frame buffer last, overwriting everything currently contained in the frame buffer in its overlapping pixel locations.

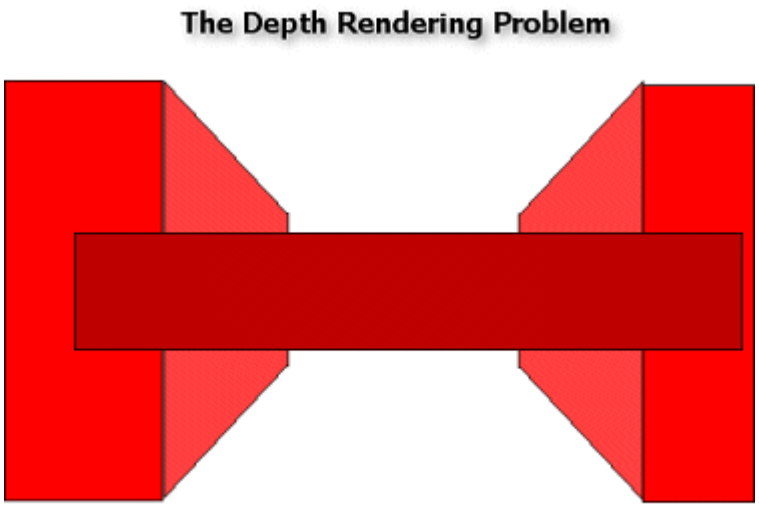


Figure 16.2

One of the early solutions to this problem was a rendering technique called the Painters Algorithm. The technique is so named because it is analogous to the way in which a painter builds up their scene on a canvas. Normally, a painter paints the background objects first and then later adds foreground objects. This ensures that the objects in the foreground are painted on top of the background objects, thus occluding them from view. The Painters Algorithm is a technique in which the polygons that are to be rendered are first sorted into a list based on their distance from the camera. The further a polygon is from the camera, the higher up in the polygon list it would be stored and the earlier it would get rendered to the frame buffer.

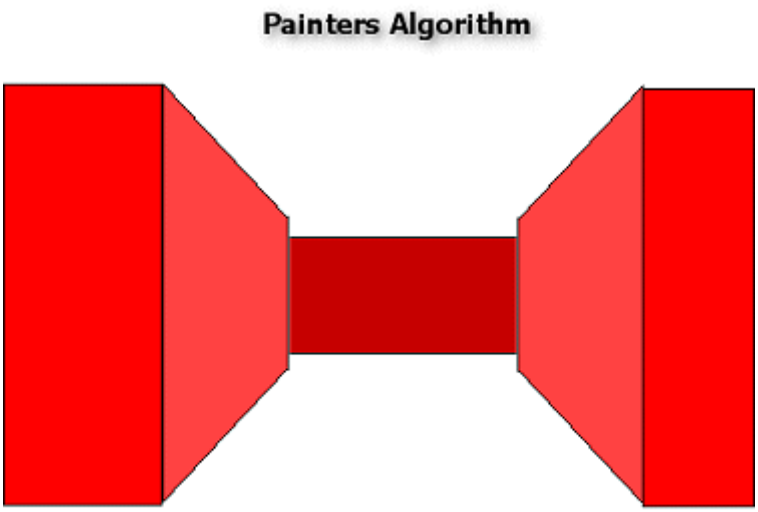


Figure 16.3

When the sorted polygon list was rendered from beginning to end, the polygons would be rendered in back to front order with respect to the camera. Nearer polygons that share the same screen space region as further polygons would automatically overwrite them, providing the correct depth based polygon occlusion.

In Figure 16.3 the same section of corridor is depicted as before, only this time the polygons have been distance sorted prior to being rendered. We can see that the first polygon to be rendered would now be the wall at the far end of the corridor. Since this one is rendered first, when the left and right wall sections are finally rendered, they automatically overwrite the correct sections of the far wall, giving the proper sense of depth.

So we see that in order to render a 3D scene correctly when no depth buffer support is available, we must sort our polygon data such that it is rendered in a back to front order with respect to the camera's current position. Keep in mind that the CPUs on these older machines were already constrained with just the actual rendering functionality; having to perform a back to front polygon sort before drawing each frame was certainly no help performance-wise. Of course, there are ways to improve depth sorting speed and efficiency. The hash table technique we implemented in Module I of this series was efficient enough, but it was far from being a perfect polygon sorter. The problem was not the technique itself, but the way that the distance to each polygon was originally calculated before being stored in that table.

Recall that in Chapter 7 of Module I, in order to sort the polygons (whether using a hash table or any other sorting technique) we had to use a point on that polygon as the sorting key. A common practice is to use the view space center position of the polygon such that the final list/table compiled describes a list of polygons sorted by the distance to their center positions from the camera. While using such a technique works quite well in a lot of circumstances, there are certain geometry configurations in which this technique will produce an erroneous render order (see Figure 16.4).

In Figure 16.4 we can see two polygons with a camera position represented by the sphere at the bottom of the image. The smaller polygon clearly overlaps the larger one from the viewer's perspective and therefore should be rendered into the frame buffer after the larger one. However, note the two arrows and their lengths representing the distance from the camera's position to the center of each polygon. We can clearly see that the camera is located closer to the center position of the larger polygon, even though the smaller polygon is in front of it. This means the method of sorting using the center point of each polygon would fail in this case and the smaller polygon would be rendered first, only to be partially overwritten by the larger polygon that is rendered afterwards. In this situation, the viewer would see depth artifacts.

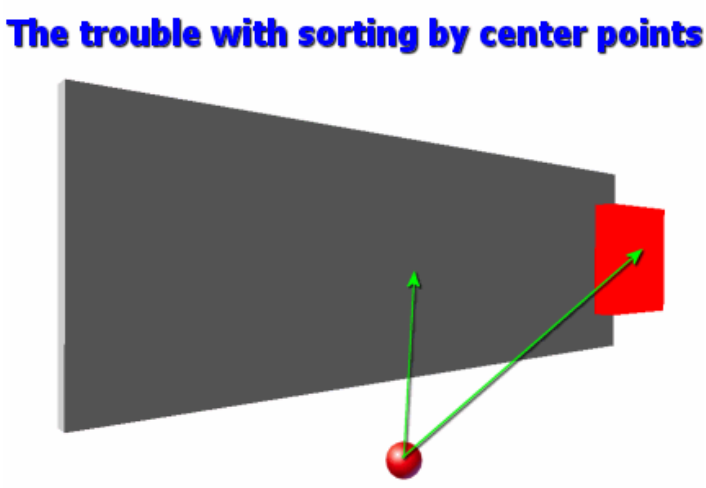


Figure 16.4

Even if a more thorough method was used to more accurately calculate the distance from the camera to the polygon and we could sidestep this particular problem, there are still certain geometric configurations for which a definitive draw order can not be ascertained. The configuration of the two polygon construct depicted in Figure 16.5 is so effortlessly handled by a depth buffer algorithm that it is easy to take depth buffers for granted.

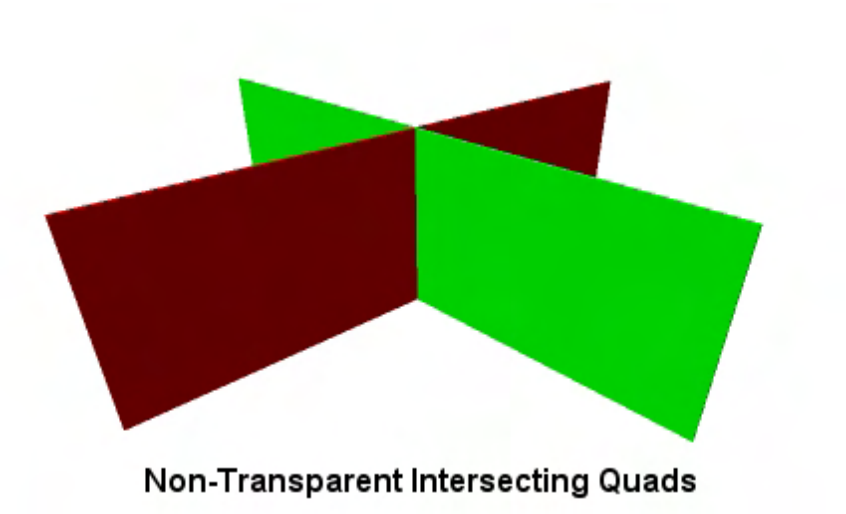


Figure 16.5

In this image we have two polygons that intersect each other to form an X shaped configuration when viewed from the camera's current location. The dark red polygon that starts at the bottom left of the image is both in front of and behind the green polygon that starts at the bottom right of the image. However, the green polygon is also both in front of and behind the red polygon.

With a depth buffer, the order in which these polygons are rendered is insignificant. If we render the red (right) polygon first, then the depth values for each of its pixels will be written the depth buffer first. When the green polygon is rendered, some of its pixels will have greater distances than those currently containing the red polygon's pixel and as such these pixels will not be rendered. We can see this in the top left of the image where, even though the green polygon was rendered second, some of its pixels are never rendered because they are found to lay behind some of the pixels of the red polygon. In the bottom right section of the image however, where we can see that the green pixels would have smaller depth values than the red pixels already contained at that location, the

red pixels are correctly overwritten by the green pixels. We have a situation where the two polygons both occlude and are occluded by each other. The depth buffer handles this case perfectly.

Looking at the above diagram one might wonder how such an arrangement could be rendered if the depth buffer was not at our disposal. Which polygon should be rendered first? As it happens, there is no solution to this problem using only polygon sorting. If the green polygon was rendered first then when the red polygon was rendered, it would overlap all the pixels in the green polygon sharing the same screen space locations, even if the red pixels were further away from the view point than the green pixels already drawn there. If the red polygon was rendered first, the same would happen in reverse and part of the red polygon would be overwritten by the green polygon in areas that should have been occluded by it. The only way this situation can be solved is by splitting the polygons into multiple sections such that the ambiguity is removed and a clear render list can be compiled for those polygon segments. This is one of the things that a BSP tree does when it compiles its data; it makes sure that when the tree is finally built, no situations like the one shown above exist in the geometry. Thus, a clear back to front rendering order can always be known at runtime.

16.2.1 When Depth Buffers Will Not Suffice

So far we have discussed the problems that faced the game engine developers of yesteryear when a depth buffer was not available for perfect depth sorting. While this discussion may have seemed nothing more than nostalgic now that we always have access to depth buffers, this is far from the case. In Chapter 7 we learned that we will need to render our alpha (i.e., transparent) polygons in a back to front order so that blending operations are carried out correctly. If we simply throw alpha polygons at the pipeline in an arbitrary order, artifacts can result since many common blending modes are order dependant.

Because we want transparent polygons to allow what is behind them to show through, we will want to render them after the main scene has been rendered. This way, the alpha polygon does not get its depth values written to the depth buffer and prevent geometry that is behind it (but should be seen through it) from being rendered. If we are looking at the outside of a house model, we should be able to see through its windows and into the room(s) on the inside. If the window was rendered before the room behind it and depth writing was being used to record the depth values of the alpha polygons in the depth buffer, when we tried to render the section of the room behind the window later, it would be rejected by the depth test. Thus, we will want to render our alpha polygons after all opaque polygons have been rendered to avoid this situation.

However, even if the alpha polygons are rendered in a separate pass after the rest of the scene has be constructed in the frame buffer, this situation can still occur if alpha polygon A is in front of alpha polygon B, but rendered first. The depth values for A will be written to the depth buffer first, and when alpha polygon B is rendered later, its pixels are depth rejected and never rendered behind polygon A. Since polygon A is supposed to be transparent, we are supposed to see polygon B behind it. Therefore B’s pixels still should have been rendered. As we learned, we can stop this from happening by simply disabling Z writing when we render the alpha polygons. This will ensure that they are correctly obscured by the opaque polygon data already contained in the frame/depth buffer and that alpha polygons

(rendered with Z writing disabled) do not occlude one another and prevent each other from being rendered.

This would work fine were it not for the fact that the blending operations that we perform are often order dependant. If polygon A is a blue window and polygon B (behind it) is a red window, the red polygon should be rendered first and the blue polygon rendered second so that we see a red window that has been tinted blue. If the red window polygon was rendered second, we would have incorrect blending where the two polygons overlap (we would see a blue window tinted red). This would indicate that we are viewing the blue window through the red window instead of the other way around.

The only way to render our alpha polygons correctly is to render them in a second pass with a perfect back to front ordering. As we can no longer rely on the depth buffer, and must sort the polygons ourselves, we find ourselves with the same problem that faced the early game engine developers. How do we render a set of polygons in a perfect back to front order that gives correct blending results every time, regardless of how the polygon data is oriented? Admittedly, we only have to do this for our alpha polygons (instead of the entire scene), but even so, a solution is required.

The hash table technique accomplished most of these goals, but occasionally failed due to the fact that it used polygon center positions as its sorting criterion. In Figure 16.6 we see both the intersecting polygon problem and the blending order problem. We are using the same two polygons from Figure 16.5, but now made transparent. Compiling polygons into a back to front render list is not always possible (see Figure 16.5) and by using the same example again but with alpha polygons, we also see the blending errors that are produced.

For the set of polygons shown in Figure 16.6, there is no clear draw order. In this example we have decided that because no draw order can be determined, we will just render the red (left) polygon first and the green (right) polygon second.

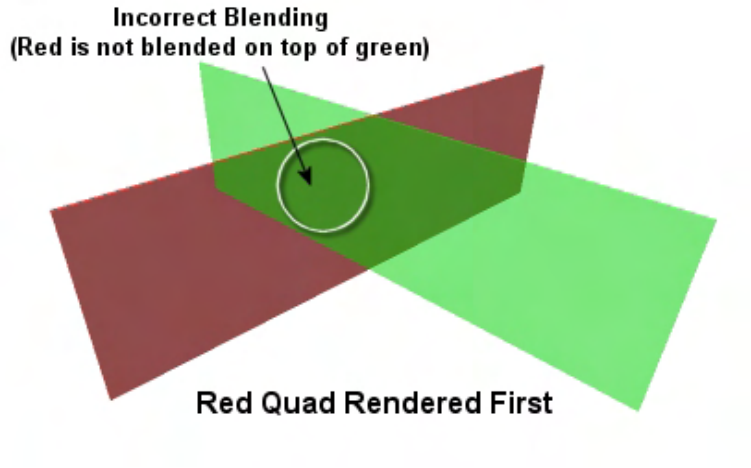


Figure 16.6

error would occur, as shown in Figure 16.7.

Because the green quad was rendered afterwards, it overwrites a portion of the red quad that is supposed to be in front of it. The rough area of green pixels that are supposed to appear to be behind the red polygon are highlighted by the circle in the diagram. Because the green polygon was rendered second, the blend order is wrong. The color of these pixels should produced by viewing a green pixel through a red polygon, but instead we are viewing red pixels through a green polygon. Of course, if we were to render the green polygon first, then the opposite

This example does not look as obviously incorrect, but after some investigation you will see that the approximate area of pixels in the region of the white circle in the red polygon should be located behind the green polygon and as such, should have been produced by blending green on top of red instead of red on top of green.

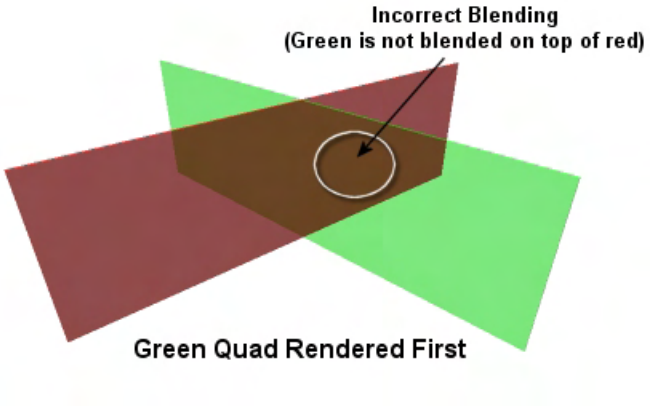


Figure 16.7

So even if we sort our polygons in a back to front rendering order so that the alpha polygons are blended in the correct order, we are still not fully covered in all cases. In a situation like this, there simply is no correct drawing order. Whichever polygon we render first, part of it will be overwritten by the second polygon in areas where it should not have been.

Figure 16.8 shows how we would like the two alpha polygons to be rendered. This is what compiling a polygon-aligned BSP node tree for our polygon data will allow us to do.

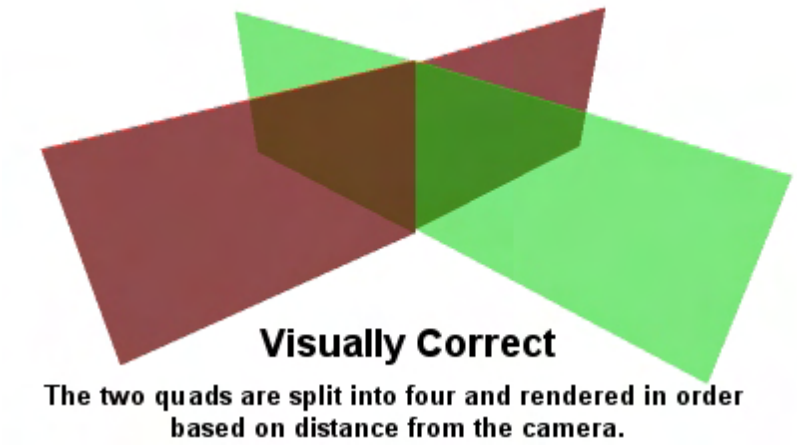


Figure 16.8

In this example, the BSP tree has used the plane of the green polygon to split the red polygon into two pieces and has used the plane of the red polygon to split the green polygon into two pieces. The original two polygons have now been replaced by four non-intersecting polygons which can be correctly ordered back to front.

In this example we can see that the back two quads (one red and one green) would be rendered first in the alpha pass. When the second two quads are rendered, the red quad is correctly blended over the back green quad and the green quad is correctly blended over the back red quad, giving us a perfect render order and perfect alpha blending.

In Lab Project 16.1, after every alpha polygon has been loaded and added to the BSP tree, the tree will be compiled. As we will discuss in the next section, this tree will allow us to render alpha data in a perfect back to front order from any camera position even though the tree is only built once. It should be

noted however that just because we are using it to sort alpha polygons, BSP trees could be used to sort the entire scene into a back to front rendering list. Indeed it was often the BSP tree that allowed the game engines of old to render polygons to the frame buffer with the correct depth occlusion without the aid of a hardware depth buffer. Thus, in the next section we will discuss building BSP trees from the more general perspective of compiling arbitrary polygon sets (not just alpha polygons).

16.2.2 Building our First BSP Tree

When we think about the spatial trees that we developed in the previous lessons, we can perceive a tree to be an arrangement of spatial relationships. Each node in the tree describes its position in the world relative to other nodes in the tree. In the case of a kD-tree for example, we know when we visit a node that the sub-tree of nodes attached to its back pointer are all representing space that has subdivided the space behind the current node's split plane. The sub-tree of nodes stored in its front list represent the space that subdivides the space in front of the current node's split plane. Thus we can think of the kD-tree as describing a collection of planes with an established relationship to one another. We know exactly which node is behind another node, and vice versa.

In order to render a list of polygons in a back to front order, we need to establish those same relationships between polygons. That is, we need to know if one polygon is behind another. Therefore, if the kD-tree tells us the relationships of planes to one another, we can use the exact same technique for our polygons. We can just use the planes of the polygons themselves as the split planes. Because we know that building a tree for these planes will describe their relationship to one another, it will therefore tell us the relationships of the polygons used to create those planes.

The process of building a BSP node tree is simple. The build procedure is passed the list of polygons that needs to be compiled into the tree. At each node, a polygon is selected from the list of polygons that made it into that node. The plane on which the selected polygon lies becomes the split plane for that node and (this is the important part) the polygon itself, and any others that may lie on the same plane, are removed from the list and stored at the node. The remaining list of polygons is then classified against the node plane and split into two polygon lists that belong in the front and back halfspace of the plane. The front list is then traversed, where it is used to create a branch of nodes in the same way as before. The back list is also traversed so that a series of back nodes are constructed, again in the same way. At the end of the process we will have compiled a BSP tree containing N nodes, where N is the number of polygons from the original dataset that lay on unique planes. Each node will store a split plane just like the kD-tree, although this split plane may not be axis aligned; it will instead be the plane of the polygon that was selected from the list at that node. Each node will also store one or more polygons that lay on that plane. Although we often refer to a tree of this type as having no leaves, what this really means is that all the polygons are not stored in the terminal nodes. If you wish, you can think of a node tree as being a tree where every node acts a leaf, because each node will store renderable data.

When the tree is built, we will be able to traverse the nodes of the tree and perform a classification test with the camera position at each node. The camera will be "pushed down the tree" so that it processes the nodes that are furthest from the viewer first. As each node is visited, the polygons stored at that node

(i.e., that lay on the node plane) are rendered. Therefore, while the tree is generated once and remains static, the order in which we visit the nodes of the tree depends on the camera position passed into the traversal function. The end result is a static tree that can be traversed in a back to front node order from any position in the scene. We will look at the rendering logic for the tree a little later; for now we will focus on the tree building process with a very simple example.

Figure 16.9 shows a level consisting of three polygons labeled A, B, and C. The camera position is not known or considered when building the BSP tree (i.e., it is view independent).

Using the example polygon set in Figure 16.9, the three polygons would be registered with our BSP tree at load time. After the polygons have been added to the tree, the BSP tree's Build function would be called to start the recursive compile process.

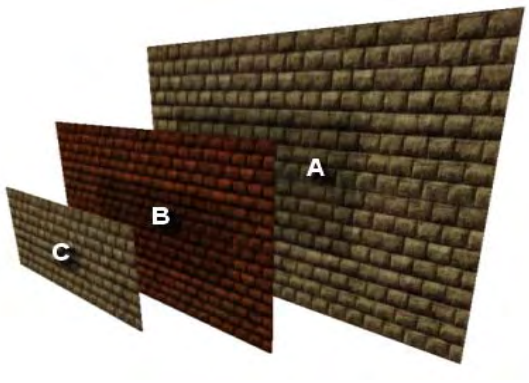


Figure 16.9

The first thing that would happen is the root node would be created and a polygon would need to be chosen from the list to use as the split plane for that node. The polygon we choose from the list at each node can be randomly selected from the list of polygons that made it into that node and the tree will still be successfully compiled (it can even be traversed successfully in the correct back to front order). However, we will learn later that there are tests we can perform at each node to find polygons which make better candidates for split planes higher up the tree, and are thus preferred. For now however, we will assume that polygon B has been chosen at random for the root node.

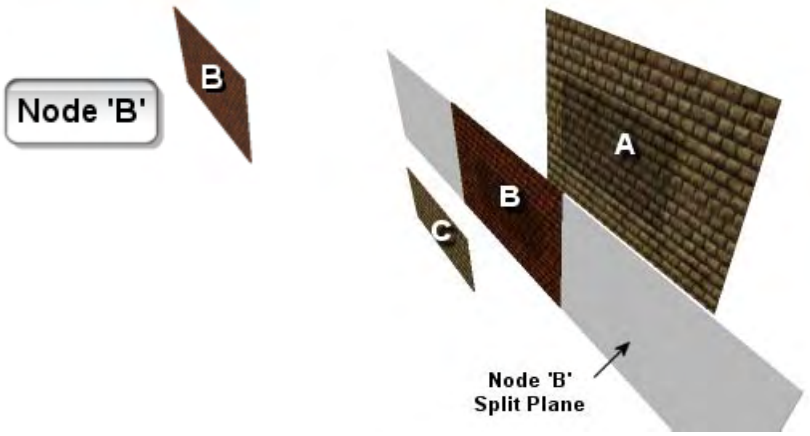


Figure 16.10

So polygon B is selected first and is removed from the list. The plane for polygon B will be calculated and stored in the root node (Node B) as the split plane. Polygon B will also be stored as the renderable data at that node.

At this point our tree has one node that stores a split plane, shown as the gray slab in Figure 16.10. Polygon B itself is also stored in the node, leaving only polygons A and C in the polygon list. This polygon list is then classified against the node plane to create two lists for both the front and back children of the root. We can see that polygon A, when classified against the split plane of node B, is found to be behind the plane and is added to the back list. Polygon C on the other hand is in front of the plane and is added to the front list. We have now done all the work we need to do at the root node. We have a split plane and a polygon stored at that node, and two polygon

lists describing the polygons that are both in front and behind the node plane. In this simple example there is currently only one polygon in the back list (A) and one polygon in the front list (C).

> **Note:** The split plane is infinitely extended in all directions, but we chose to display only the portions that match the vertical dimensions of the polygon for ease of viewing in the diagrams.

As the front and back polygon lists are not yet empty, we must recur down the front and back of the node and create additional child nodes. Two nodes are created and attached to the root node's front and back pointers. The function then recurs into each child passing in the respective list. In this example we will assume that we step into the back child first, passing in the back list which contains a single polygon (A).

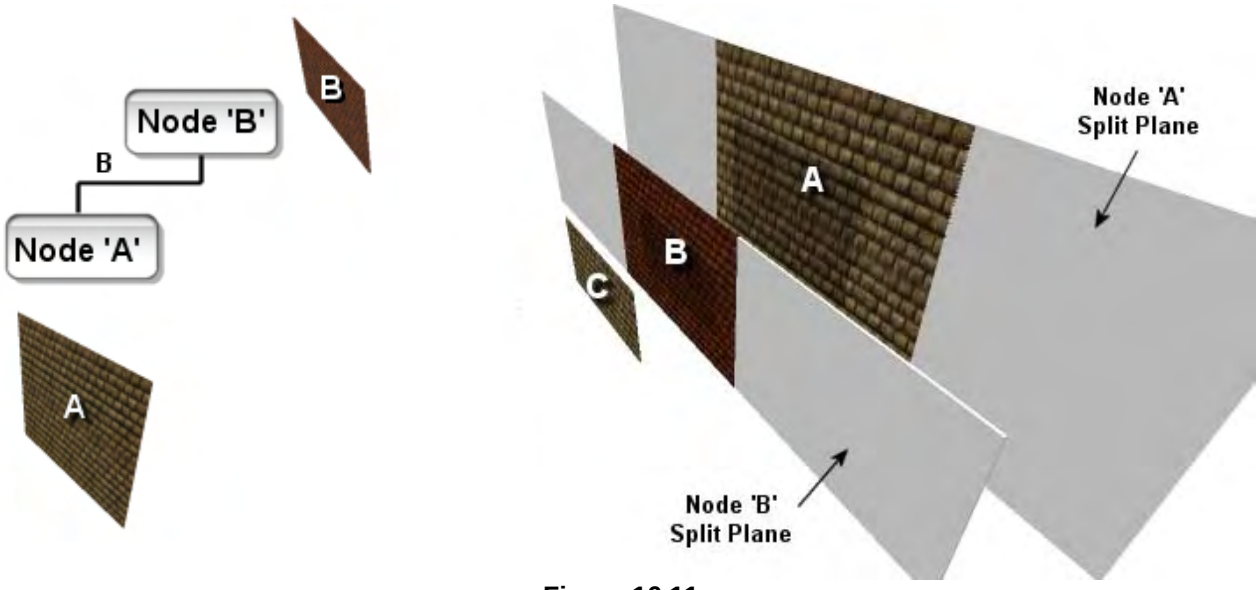


Figure 16.11

When we arrive at the back child of the root, the same process happens. A polygon has to be selected from the passed list so that its plane can be used as the split plane at that node. Since the polygon list passed into the back child contains only one polygon, Polygon A, the choice is easy. Polygon A is removed from the list and stored in the node. The polygon's plane is calculated and stored as the split plane for this node, which is called Node A in the diagram. As the polygon list is now empty, we have reached a terminal node. Because there is no polygon data left to subdivide, we no longer have to compile any front of back polygon lists for this node and we do not have to create any child nodes. With polygon A and its plane stored in the node, we return back up to the root.

At this point we have processed the root's back child but have not yet traversed into the front child with the front polygon list. This is done as the final step for this node.

As we step into the front child, we once again find ourselves at a new node that needs to have a plane selected for it from the passed polygon list. In this example, the polygon list passed into the front node contains a single polygon (C), and therefore this polygon is removed from the list and stored in the node along with its plane. As the polygon list is now empty, no child lists or nodes have to be compiled and

the function can step back up to the root. Once back at the root node, we return from the build process and hand program flow back to the application.

Figure 16.12 shows the final representation of our three polygon BSP tree after the front node has been visited and polygon C assigned to it.

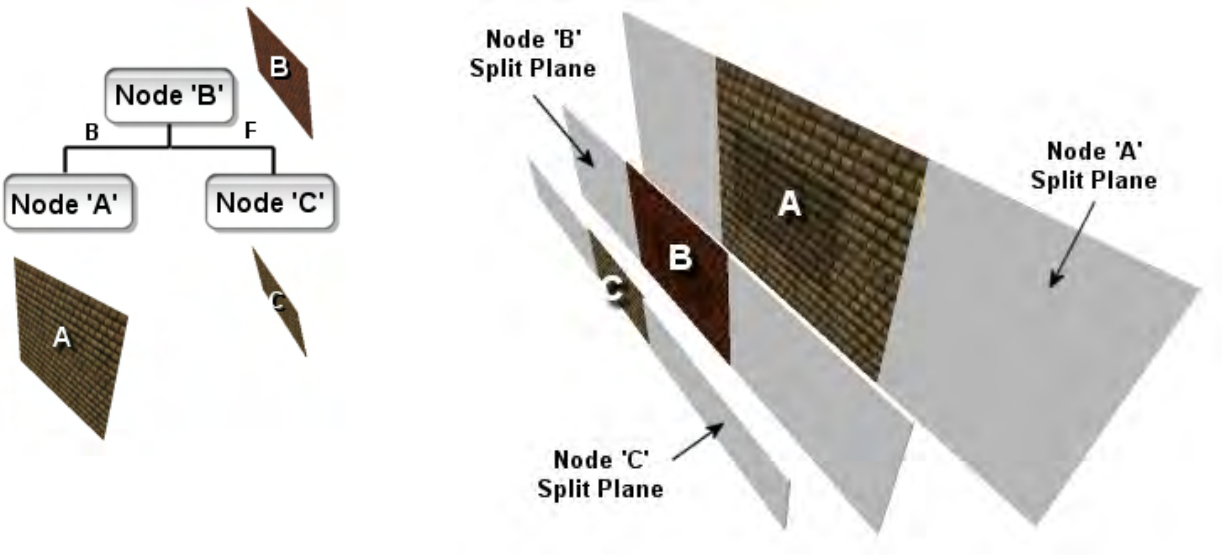


Figure 16.12

Looking at the tree diagram on the right hand side of Figure 16.2, you can see that we have created a three node tree from our list of polygons and we have stored a polygon at each node.

16.2.3 Rendering Our First BSP Tree

In this next section we will examine how to render a BSP node tree in perfect back to front order using the simple example tree discussed in the last section. We will look at two examples using two different camera positions so that we can see that while the tree is compiled once at startup and never changes, it can be traversed such that the polygon draw order is dynamically created in a back to front order from any camera position.

The rendering of the BSP node tree is a typical traversal which, as we would now expect, will be implemented using a recursive function. The application will render the tree by calling this function for the root node (passing a root node pointer and a camera position). The function will visit each node and perform a fast and simple test to determine which child should be visited first. This will depend on which halfspace of the current node's plane the camera is currently contained in.

For each node visited during the render traversal, the basic logic performed is as follows:

- Classify camera position against node plane

- If camera position is in frontspace of plane
    - Step into back child
    - Render polygon stored at the current node
    - Step into front child

- Else if camera position is in backspace of plane
    - Step into front child
    - Render polygon stored at the current node
    - Step into back child

This traversal logic at each node makes a lot of sense. We are essentially stating at each node, if the camera is in front of the plane, render the stuff behind it first, then render the polygon stored at this plane, and finally render the stuff in front of the plane. Alternatively, if the camera is behind the current node plane being tested, render everything in front of the plane first, render the polygon data stored at the node, and then finally render the data behind the plane. This traversal logic means that at each node the choice of which order in which to visit each child node is dependant on the camera position. Thus, when the entire tree has been visited, the node's polygons will have been rendered in a back to front order with respect to the position of the camera.

Imagine that the node structure for our BSP tree looks something like this:

```
struct BSPNode{
  POLYGON *splitter;
  BSPNode *FrontChild;
  BSPNode *BackChild;
};
```

Given the above, the following semi-pseudo code shows how our recursive BSP tree rendering function might look. This function would be called by the application just once each frame and passed the root node of the BSP tree and the current camera position.

```
void RenderBSP (BSPNode *CurrentNode , D3DXVECTOR3 &CameraPosition)
{
  int Result = ClassifyPoint( CameraPosition, CurrentNode->Plane );

  if (Result == Front || Result == OnPlane)
  {
      if (CurrentNode->BackChild != NULL)  RenderBSP (CurrentNode->BackChild );

      DrawPolygon(CurrentNode->Polygon);

      if (CurrentNode->FrontChild != NULL) RenderBSP (CurrentNode->FrontChild);
  }
  else
  {
      if (CurrentNode->FrontChild != NULL) RenderBSP (CurrentNode->FrontChild);
      if (CurrentNode->BackChild != NULL)  RenderBSP (CurrentNode->BackChild );
```

```
    }
}
```

In this example, the DrawPolygon function is assumed to be a function that renders the passed polygon to the frame buffer. Notice that there is no need to render the polygon data stored at the node if the camera is located in its back space. This would obviously mean the camera is looking at the back of the polygon, which would be back face culled by the pipeline anyway.

Rendering Example 1

In this first example which uses our three polygon BSP tree from the previous section we are given a camera position as seen in Figure 16.13. We can clearly see that in order for the polygons in the tree to be rendered in the correct back to front order we should render polygon A first, followed by polygon B and then finally polygon C. What is also worth noting is that the orientation of the camera is not used in the traversal logic at all, which may at first seem strange. Only the camera position is used and classified against the node planes to determine the correct drawing order.

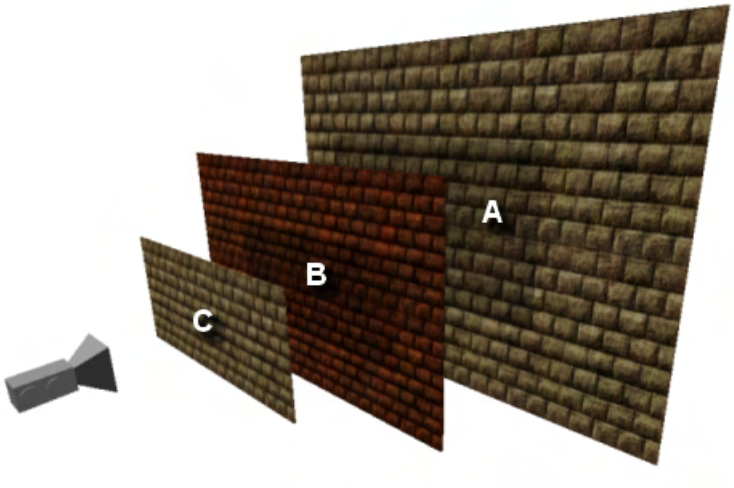


Figure 16.13

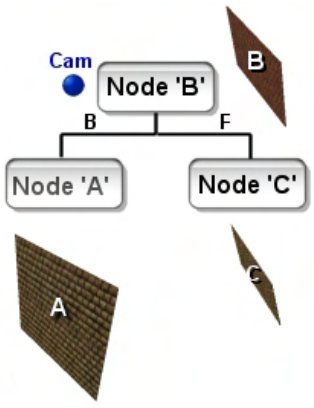


Figure 16.14

In Figure 16.14 we start our traversal of the tree by passing our camera position into the root node. This is shown as the blue dot labeled ‘Cam’ in the image. As discussed and shown in the previous pseudo code section, the first thing we do is classify the camera position against the plane stored at Node B. In Figure 16.13 we can see that the camera is currently in front of the split plane stored at Node B

Note: Remember, the split plane stored at each node is the plane that the polygon lies on. Therefore, we can see that the camera position would be in front of Node B because it is in front of polygon B, which is just a subset of the same plane.

As the camera is currently in front of node B, our logic tells us that we must visit the back child first before rendering the polygon stored at this node. So in the next step, we traverse into the back child of node B and arrive at Node A.

As Figure 16.15 shows, when the camera arrives at node A, it is once again classified against the split plane stored there and we determine that it is in front of node A. This would normally mean that we

would step into the back child first before rendering the polygon stored at that node. However, as no back list exists, we skip that part and simply render the polygon stored in the node (A).

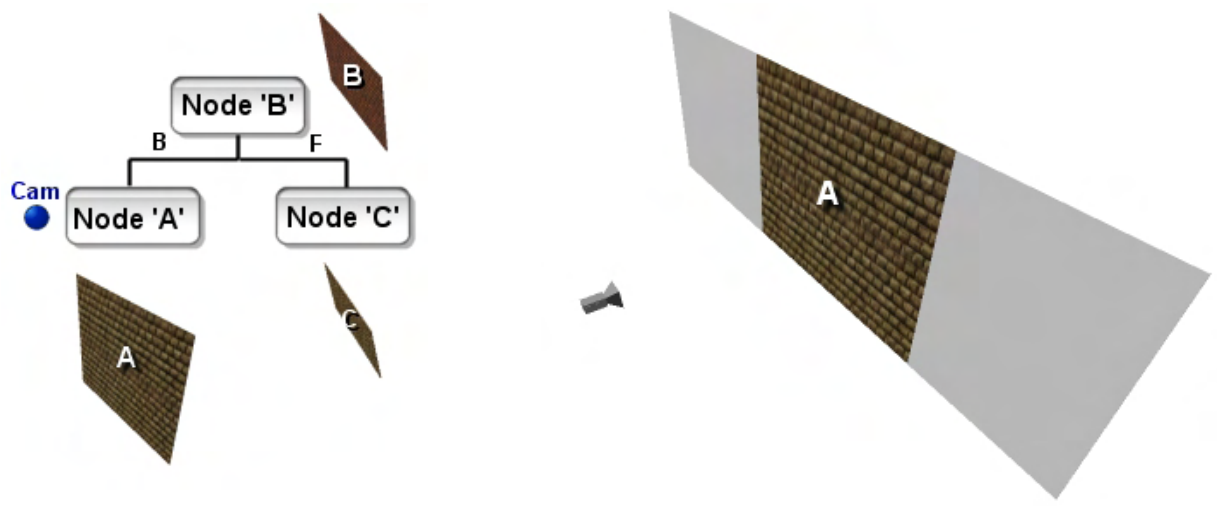


Figure 16.15

After rendering the polygon at node A we would step into the front child next, but this node is a terminal node that has no front list, so we return from the current instance of the recursive function and pop back up to the root node just after the point where the back child was stepped into.

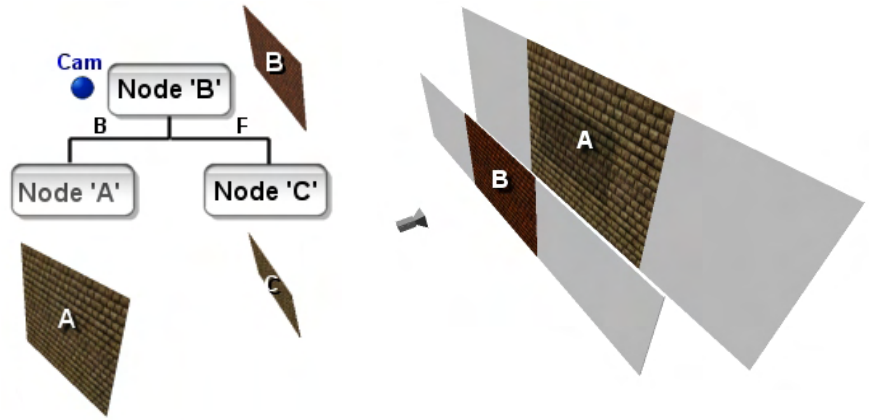


Figure 16.16

Before we stepped into the back child of the root node, we determined that the camera position was in front of the root node. This determined that we had to render the back child first, then render the polygon stored at the node and then traverse into the front child. We have now completed the first step and have stepped into and have returned from the back child. Therefore, our next job is to render the polygon stored at the node. The polygon stored at the root node B is polygon B, so we render this polygon next, as shown in Figure 16.16. Since this polygon is closer to the camera than polygon A, some sections of polygon A will be overwritten by polygon B in the frame buffer. This is absolutely correct because, from the camera's view point, polygon B should partly occlude polygon A.

We have now performed two of the three steps we had to perform at the root node (we rendered the back list and rendered the polygon stored at the node). All that is left to do now is step into the front child of the root node, where we visit node C.

Note: Remember that the order in which we decide to traverse into the front and back lists is dependant on the camera position. If we are in front of the plane, we step into the back child first. If we are behind the plane, we step into the front child first. This ensures that we always visit the nodes first that are in the opposite halfspace of the current node's split plane with respect to the camera position.

As Figure 16.17 shows, when we visit the front child we find that it has no children, so we simply render the polygon stored at the node (i.e., polygon C). As polygon C was rendered last, it will overwrite sections of polygon B in the frame buffer, which again is absolute correct. Polygon C is closer to the camera and therefore should occlude polygon B to some extent.

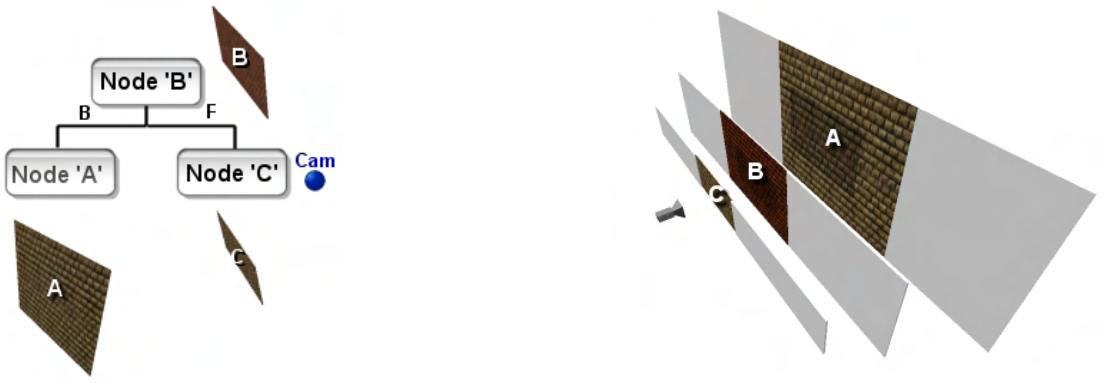


Figure 16.17

After polygon C is rendered, we return from node C back up to the root. At this point we learn that the root node has performed all its tasks, so the root node call returns and program flow passes back to the application. Although this was a simple example, we have just seen how to use a BSP tree to render our small scene in back to front order. Again, the same function can be used to render the same tree in a perfect back to front order even when the camera position changes. Stepping through one more rendering traversal using the same BSP tree with a different camera location will solidify our understanding of the traversal process.

Render Example 2

In this second example we will use the same BSP tree, but move the camera such that it is situated between polygons B and A as shown in Figure 16.18. Once again you should take note that although the camera in Figure 16.18 is facing in a given direction (in fact, almost opposite the earlier orientation), this is not factored during the traversal. Rendering order is determined using only the camera position.

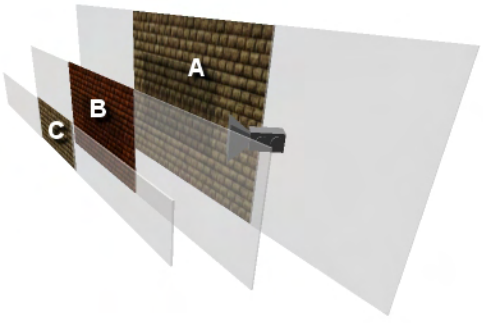


Figure 16.18

As before, we start off with nothing rendered and send the camera position into the root node to begin the traversal process.

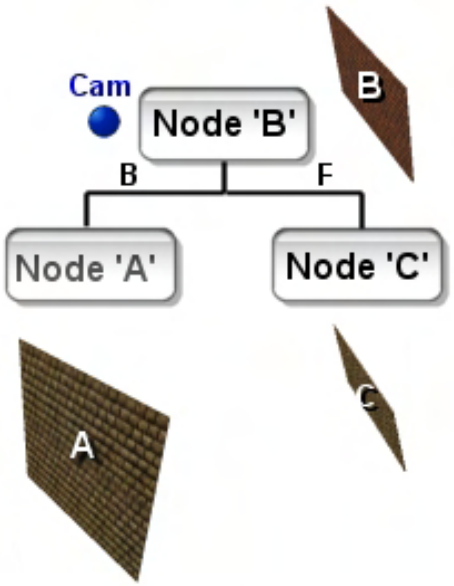


Figure 16.19

At the root node we classify the camera position against the plane of polygon B and find that it is located in the plane's backspace. This differs from our previous example where the camera was located in the root node's frontspace. As such, the order in which we traverse into the front and back children must be reversed. Because the camera is now located in the backspace of plane B we know that the more distant polygons must be located in the frontspace of B, and therefore should be visited and rendered first.

The order in which we do things at the root node will now be to step into the front child first and then render the polygon stored at the node (polygon B) before finally stepping into the back child. As we are currently at the root node, our first task is to step into the front node (Node C).

At Node C we find that we are visiting a terminal node and as such, there is no front or back child to traverse into. Therefore we just render the polygon stored there (see Figure 16.20). However, because the camera is behind the polygon it would be back face culled, so we would normally only render the polygon data stored at any node if the camera is in the plane's frontspace. But in this particular example, we will go ahead and render it for the purposes of demonstrating the correct traversal order being executed. At this point we have rendered C as our first polygon and when the function returns, we pop back up to the root node.

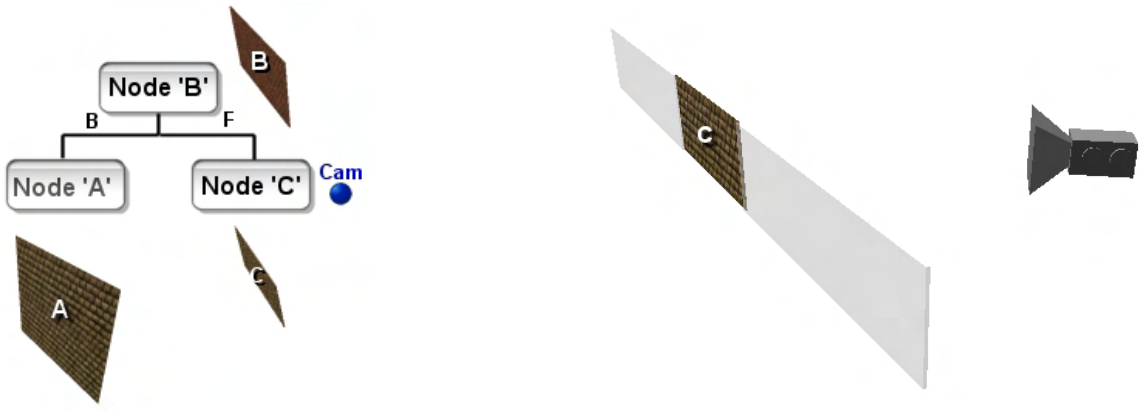


Figure 16.20

Once we are back at the root node, having traversed and (potentially) rendered the front tree of Node B (which in this example contains just Node C), we then render the polygon stored at Node B. However, once again, as the camera is in the backspace of node B, there is actually no need to render the data stored there as it would ultimately be back face culled. This means polygon B becomes the second candidate for rendering this time around (see Figure 16.21). Again, for this particular example, we will render the polygon anyway (at least in our images) so that you can see the order in which the polygons would be rendered if they were actually facing the camera.

Figure 16.21

Having rendered the polygon at the root node, we then step into its back child where we arrive at terminal Node A. This node then has its polygon rendered, as shown in Figure 16.22.

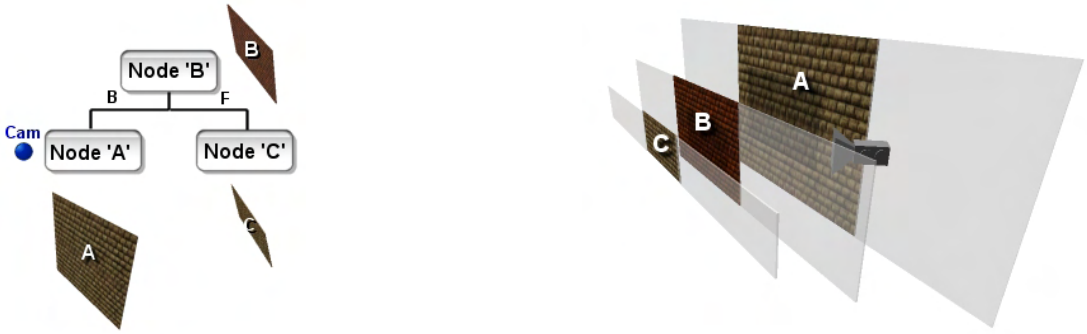


Figure 16.22

Just remember that in reality, polygon A would be the only polygon rendered in this scene as it is the only one in which the camera is located in its front space. At this point the polygon has been rendered and we step back up to the root node and return program flow back to the application.

We have now seen two examples of rendering the same BSP tree using different camera locations. It should be clear to you that even though the tree was only compiled once (hopefully in an offline process, although that would not be necessary for our simple example), it can be dynamically traversed at run time to generate a perfect back to front drawing order for polygons.

16.2.4 Perfect Back to Front Ordering with BSP trees.

In the previous section we examined a very simple example of a three polygon scene compiled into a BSP tree which we rendered using a specific set of traversal logic to determine a back to front draw order. Of course, the example BSP tree we used was deliberately simplistic to teach the fundamentals of BSP tree construction and traversal. All we have really done so far is introduce another technique to essentially give us the same results as our hash table (or any other sorting technique). After all, the polygons did not intersect each other and cause the ambiguous polygon order scenario. If our above example had contained such polygons, the technique we discussed would have failed since the same

problems still exist. So what if we were to discover a situation during the build process where polygon A is both behind and in front of polygon B? Should it be added to the node as a back child, a front child, or split by the node plane and have each fragment added to the relevant lists? The answer is that we must split polygons that span planes during compilation. Indeed it is this very concept that solves the geometric ambiguity issue for intersecting polygons.

In order to understand the full BSP tree building process, we will use a slightly more complex set of data that causes some spanning polygons that will need to be split during the build process.

In Figure 16.23 we see five polygons labeled A through E. We will assume that the camera is looking at the front of all the polygons here. That is, the polygon normals all face in the general direction of the camera. Looking at this collection of polygons, we have to try to distinguish which polygons should be rendered first and which should be rendered last based on the camera position.

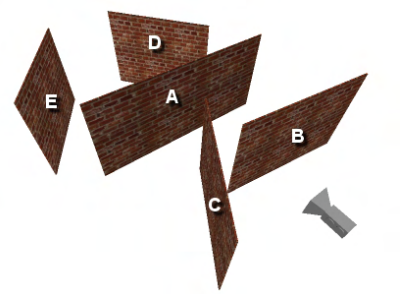


Figure 16.23

We can easily see that the first two polygons to be rendered should be polygons E and D, followed by polygon A. But which polygon should be rendered last, C or B? C is spanning the plane on which polygon B lies and is therefore both in front and behind polygon B. The order in which to render these polygons is now unclear and cannot be immediately determined. We can also imagine that such a geometric arrangement would be problematic in the build process of the tree using the techniques described above. Imagine for example that polygon B was chosen as the root node of the tree. We know that we then must classify the other polygons against this node plane to send them into the front or back list. We can see that polygons A, D and E would all be assigned to the back list of B but which list would polygon C belong to if it exists in both the front and back halfspace of node B's plane?

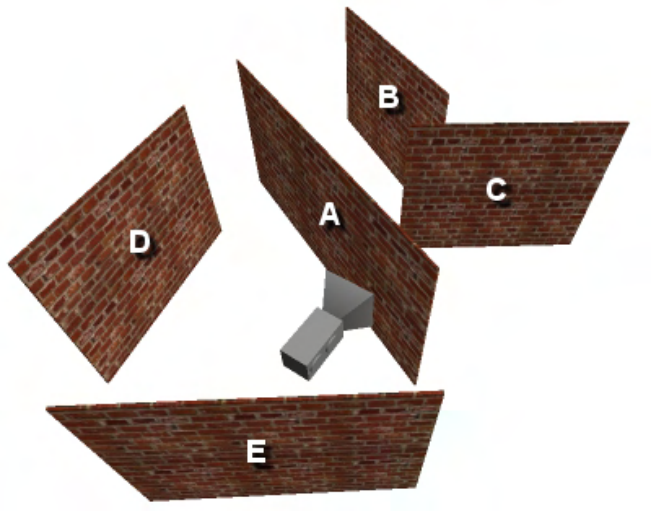


Figure 16.24

Figure 16.24 shows the same scene with the camera in a different position. This arrangement more clearly shows the problems that occur when one polygon spans the plane of another polygon.

Imagine what would happen if polygon C was chosen to create the split plane at the root node. Both polygons A and D would be spanning this plane, and this is not something we can allow to happen. For the BSP tree to provide a correct back to front polygon collection routine, every polygon must be either in front, behind, or on plane with another polygon. As soon as we have a polygon that spans a node plane anywhere in the tree, we lose the relationship between the polygons that tell us precisely

which polygon must be visited first.

The solution to this problem is simple. When a node is created, any polygons remaining in the list that made it into the node are classified against the node plane to decide whether they belong in the front or back list. Any polygon that is spanning the node is split, creating two new polygons which replace the original polygon in the list. Essentially, any polygon that spans a node plane, and is therefore causing us a problem, is split into smaller components that can be accurately described as belonging in either the back list or the front list of that node.

Without further ado, let us look at an example of building a polygon-aligned BSP node tree with the splitting logic introduced. With splitting in place, our BSP tree will be able to compile any geometry and render that geometry in a perfect back to front order from any camera position in real-time.

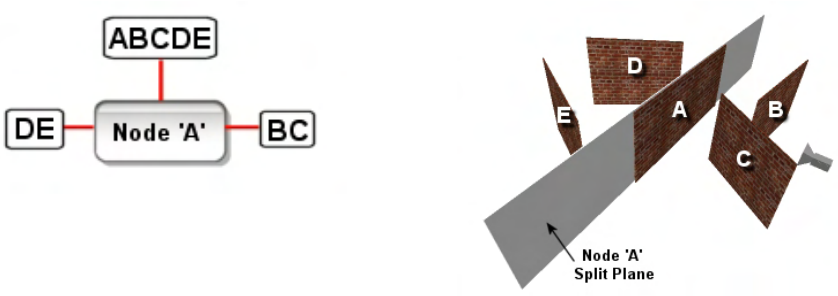


Figure 16.25

In this example we will use the same five polygons shown in the previous diagrams. For the sake of demonstration we will choose the split planes at each node in alphabetical order, but as discussed, the BSP tree will build and work correctly choosing the splitters in any order.

In the first step shown in Figure 16.25 the recursive build function is passed a root node and a list of five polygons labeled A through E. In this example, polygon A is chosen to be the splitter at the root node and is therefore removed from the list and stored in the node. In Figure 16.25 we have depicted the plane as the gray slab extending out from polygon A. As we can see, the root node now splits the space of the scene into two nearly equal sub-spaces. Polygon A is now stored at the node and the polygon list contains polygons B through E. In the next step, these polygons are classified against the plane of polygon A in order to construct front and back lists that will be passed down to the respective child nodes. If we look at Figure 16.25, we can see that D and E get assigned to the back list and polygons B and C get assigned to the front list. As we have non-empty front and back lists at the root node, it means we must create front and back child nodes and attach them to the root.

In this example we will recur into the front node first where the input list includes polygons B and C. A new split plane has to be chosen at the child node and, because we are going alphabetically, polygon B is selected from the list. The split plane is generated and stored in the node, along with polygon B. With polygon B now extracted from the list and stored in the node, the polygon list at node B now contains only one polygon: polygon C

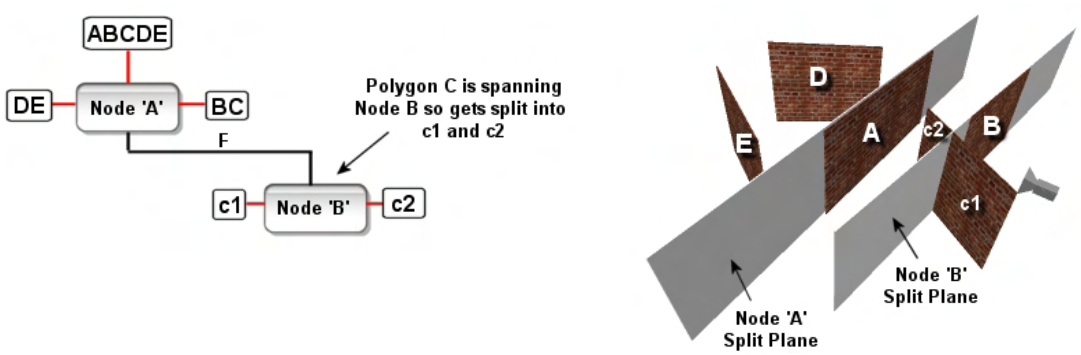


Figure 16.26

.

At this point, our job is to classify the remaining polygons in the list against the node's plane. In this instance, the polygon list contains just one polygon, so we would classify polygon C against the plane of Node B to determine whether it should be sent to the back or front of the node.

When polygon C is classified against the plane it is found to be spanning that plane and therefore polygon C must be broken into two child polygons. When we split a polygon against the plane of a node, the clipped fragment that is situated behind the plane is added to the node's back list and the fragment in the node's frontspace is added to the front list. After splitting polygon C into two child polygons (c1 and c2) and deleting the original polygon, we now have a front list and a back list compiled at node B. Each list contains a single polygon. The back list at this point would contain polygon c2 and the front list would contain polygon c1 (see Figure 16.26).

Because Node B still has polygons in its front and back lists, we will need to create both front and back child nodes. We then have to step into each of these children to continue the recursive building process. We will step down into the front child of Node B first.

When we enter the front child of Node B we are passed a polygon list that contains the single polygon c1. This final polygon is selected as the splitter at the new node, creating the Node c1 shown in Figure 16.27. At this point there are no more polygons in the list, so we return from Node c1.

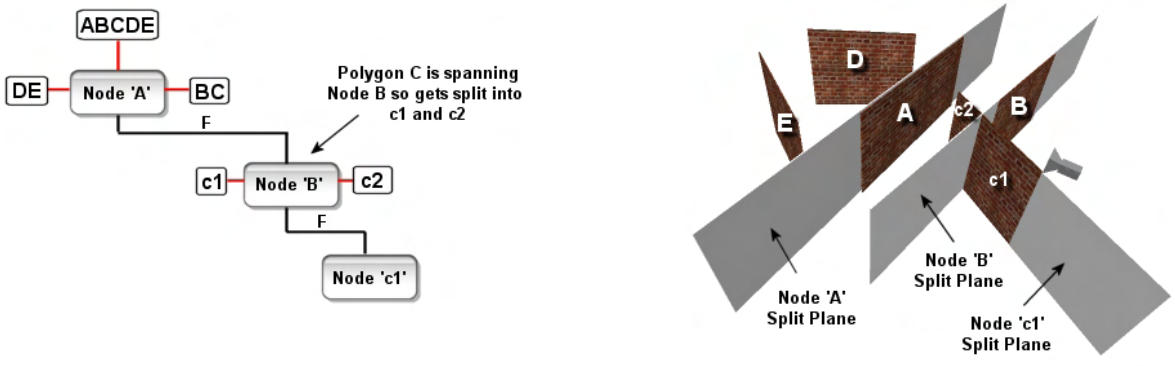


Figure 16.27

After we return from node c1 we find ourselves back at node B having built its front branch. As this node also has a polygon still in its back list (c2), we step into the back child of node B next and build its back tree.

As with node c1, when we step into the back child of B, we are passed a single polygon (c2), so the polygon to use as the split plane at this node is obvious. Polygon c2 is removed from the list and stored in the node labeled c2 in Figure 16.28

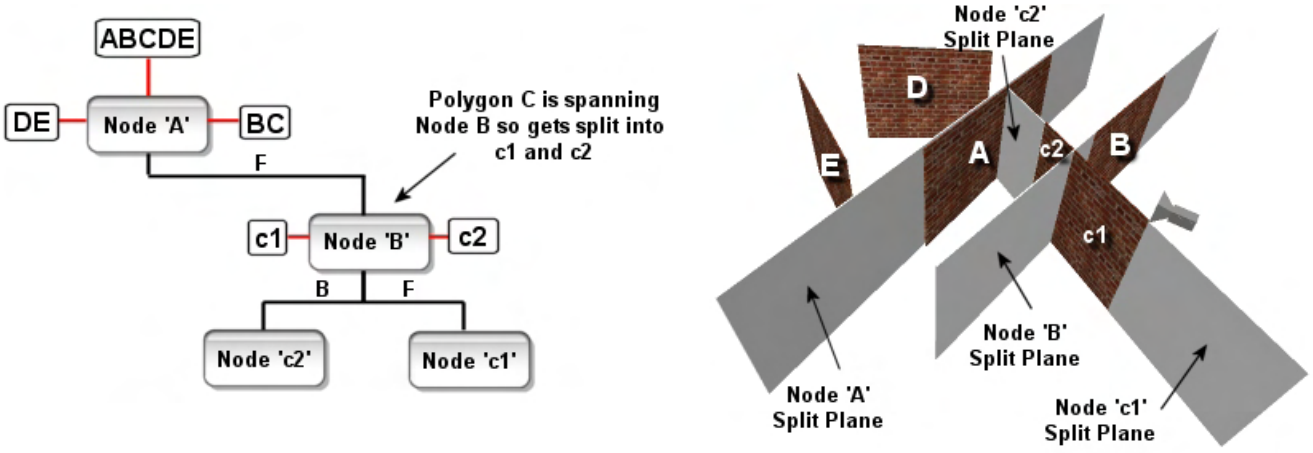


Figure 16.28

After storing the split plane and polygon at Node c2 we know we have reached a terminal node because there are no polygons remaining in the input list. In this case, the recursive procedure returns and program flow steps back up to Node B. At Node B we also realize that we have built its front and back sub-trees at this point, so there is no more work left to do at this node either. Thus, we return from Node B, popping up the tree and back to the root node A. At Node A we have only processed its front sub-tree, so it is now time to recur into its back child with the back list that was compiled at this node. The back list compiled for Node A contained polygons D and E, so these are the polygons that are passed into the newly created back child node of the root. When we step into the back child we are passed polygons D and E and one of them must be removed from the list and used as a splitter. In keeping with our alphabetical scheme, we will assume that polygon D gets selected and stored in the node as shown in Figure 16.29

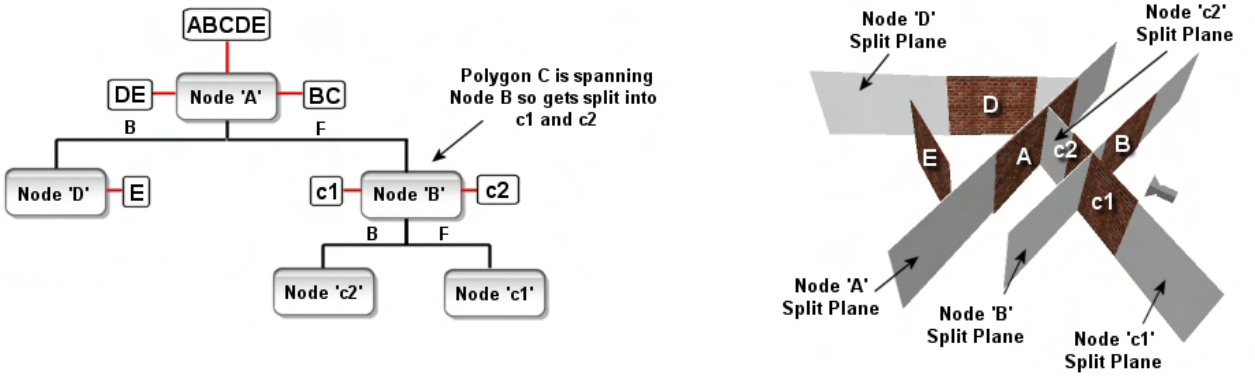


Figure 16.29

After the split plane for polygon D has been stored in the new node (called Node D), the remaining polygons in the list are classified into front and back lists. In this example, only polygon E remains in the list. Since it is found to be in front of Node D, it will be added to the front list. As there is no back

list compiled for Node E we do not have to create a back child node. However, there is still a single polygon in its front list, so we must create a new front child node and step into it.

In the front child of Node D we find only polygon E as its input data, so this becomes the splitter for the final node down this branch of the tree (see Figure 16.30).

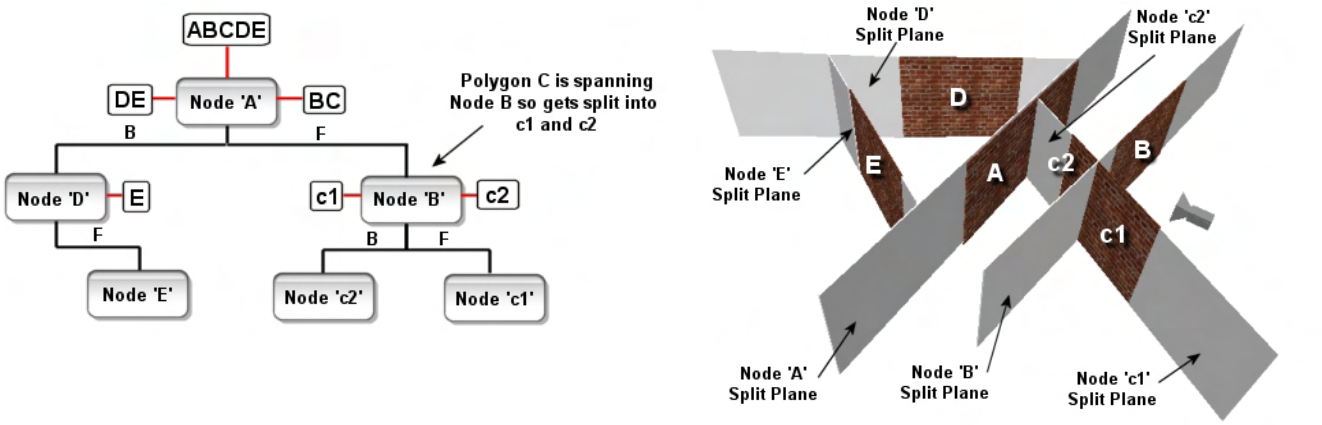


Figure 16.32

We return from Node E back to Node D where, after finding no back child has been created for this node, our task is done and we step back to Node A. When we unwind the call stack all the way back up the back branch of the tree and arrive back at the root node, we find that we have created the front and back trees of the root node and thus built the entire tree. At this point we return from the root node and the build process is complete.

16.2.5 An Overview of the Build Procedure

We have now built a BSP tree that has correctly split overlapping and intersecting polygons so that no rendering order ambiguity exists. It also worth noting that in our example the geometry set did not contain any polygons that shared planes. That is, we did not encounter a situation such that when classifying the polygon list against a node plane, we found a polygon that shares the plane with the node.

In such situations there are several things that can be done and they will all work with varying degrees of efficiency. Below we discuss possible strategies for dealing with the on plane case during the polygon/node classification step.

- You could test the normal of the polygon and add it to the front or back list depending on whether its normal is facing in the same direction as the node plane's normal or not. This would mean that each of the polygons that share the plane will have their own nodes created in the tree. Although this will work fine and still produce a perfectly valid tree, it does mean that we will have several nodes in the tree that essentially describe the same plane. As the BSP tree is being used to ascertain depth order based on view space plane distance, this redundancy is a little pointless. We know for example, that all polygons that share the same plane can be rendered

together since they cannot be occluding each other (they all describe the same depth slice of the scene).

- We could store any polygon that exists on the same plane as the node plane in the node itself, even if the normal of the polygon faces into the opposing halfspace. That way, even if 100 polygons spread throughout the level existed on the same plane (even if facing in opposing directions) only one node would be added to the tree and all 100 polygons would be stored at this node. The problem with this approach is that, because we now have multiple polygons stored at a node which may be facing in opposing directions, we must always render the polygons stored at that node even if the camera position is located in the plane's backspace. We mentioned earlier that during the rendering traversal we only render the data stored at the node if the camera is in front of the node and can thus potentially see the front side of the polygon stored there. However, with this technique, even if the camera is located in the backspace of a node plane, this does not mean that it will not contain polygons that are facing into the backspace. Therefore, we will have to render the polygon list stored there, whether the camera is behind or in front of the plane. This could mean that at a given node we render many back facing polygons that will be back face culled by the pipeline only after they have been needlessly transformed.

- The third option is to use a slight modification of the second strategy discussed above. That is, when we select a polygon whose plane is to become the node splitter, we find any remaining polygons in the list that share the same plane (even if facing in opposing directions) and store them at the node. Where this differs from the previous strategy is that the node structure itself will now have two polygon lists. One list will contain on plane polygons that face in the same direction as the plane and the other will contain the on plane polygons that were found to be facing into its backspace. With this technique, we get the best of both worlds with only a slight adjustment to our rendering strategy. Now, when the camera is located in the frontspace of the plane we only render the list at that node containing the polygons that face in the same direction as the node plane normal. If the camera is located in the backspace, we render only the list of polygons stored at the node that have opposing face normals. This technique is desirable because it means we create only one node in the tree for a given plane even if there are hundreds of polygons located on that plane. The result is a shallower tree that is more efficient to traverse. By storing many polygons at the nodes, we also have the ability to render them with a single call, speeding up rendering. Finally, as we have two lists of polygons stored at the node (of which only one is ever rendered depending on the camera's location with respect to the plane), we automatically perform back face culling with zero overhead.

- This final strategy is a mix of the ones described above and is the one we are going to implement in Lab Project 16.1's BSP tree compiler. We go back to the concept of a node storing just a single polygon list that is only rendered when the camera is in the node's front space. When a node is created and a polygon is removed from the list to create its plane, we will also remove any polygons that share the same plane and have same facing normals. We will store them at the node as discussed. Polygons that are found to be on plane with the node having opposing normals will not be stored at that node -- they will be added to the back list. This means, they will be used later to create additional nodes. That is, a node will contain a list of polygons that share the same plane and are facing in the same direction (the node's front space). These

polygons can all be rendered together when the node is visited and is found to have the camera in its front space.

One might wonder why we decided to opt for the fourth strategy instead of the third, when it seems to create more nodes. In truth, while both techniques provide good performance, our decision was slightly biased because the final strategy is the one that we ***must*** use during the building of a BSP leaf tree (discussed in the next section). Rather than having two different build strategies for the leaf and node trees, we felt that it would be better to unify the processing of the on plane case across BSP tree types. This will also help our transition into the BSP leaf tree discussion.

The following notes demonstrate how the BSP tree we build in Lab Project 16.1 will construct a node.

- Enter Node N.
- Select a polygon from the input list and store its plane at the node.
- For each polygon in the list.
    - Classify polygon against node plane
        - Front : Add to front list.
        - Back : Add to back list.
        - Spanning : Split polygon and add each fragment to front and back lists.
        - On Plane
            - Same facing normal : Store at node.
            - Opposing facing normal : Add to back list.
- If (Front list)
    - Create new front node and attach to current node.
    - Recur into front node passing in the front list of polygons.

- If (Back list)
    - Create new back node for the current node.
    - Recur into back node passing in back list of polygons.
- Return.

If you look at the CBSPNodeTree::Build function in the CBSPNodeTree.cpp source file that ships with Lab Project 16.1, you will see that these are the exact steps taken to construct each node. The source code to the entire class will be explained in the accompanying workbook

We now know how to compile a BSP node tree, and in the last build example we used a slightly more complex set of geometry which had intersecting polygons (or polygons with intersecting planes). The splitting process that occurs in the BSP tree build procedure resolved any ambiguity by clipping polygon C to the plane at Node B, thus creating two separate polygons which no longer span the plane. Let us now look at one final rendering example using the example BSP tree we have just built so that we are sure that we fully understand why the clipping of polygon C during the build process means that we can now calculate a perfect draw order.

Render Example 3

In this example we are using the tree that we used in the last build example. We start off by visiting the root node with the camera. To the left in Figure 16.33 we see our currently compiled BSP tree. The blue dot in this image represents the location of the node we are currently visiting as we step through the traversal. As you can see, the blue dot is currently at the root node since this is the first node we will visit with the camera. Although nothing has been rendered yet, the right hand side of Figure 16.33 shows the node planes that the tree represents and the way in which the space has been carved up. We can also see the location of our camera as currently being in front of nodes B and c1. All node planes in this example are assumed to be oriented in the general direction of the camera. It is this camera location we are now going to traverse the tree with to determine the correct draw order.

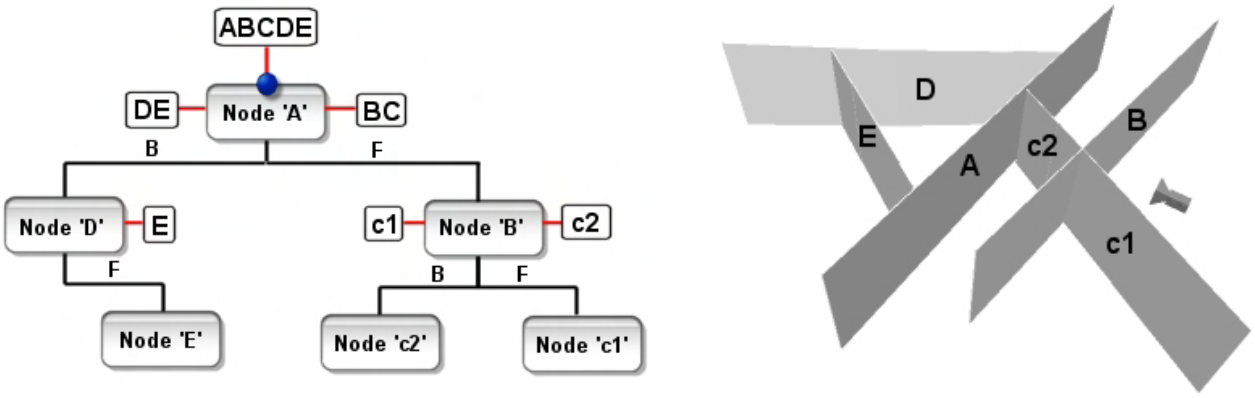


Figure 16.33

At the root node, the camera position is classified against node plane A and is found to be contained in its frontspace. This means we must step down the back of Node A first. When we step down the back of Node A we arrive at Node D (see Figure 16.34).

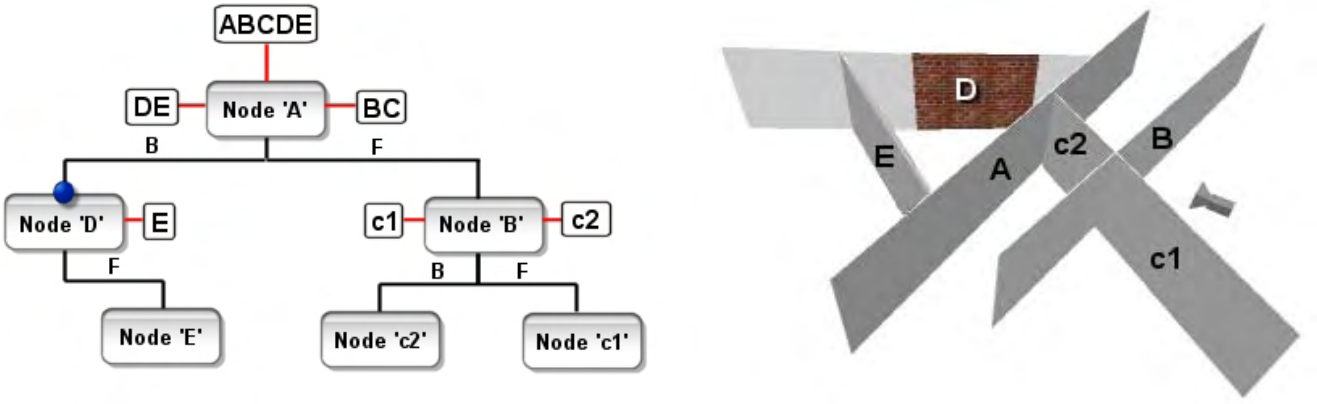


Figure 16.34

At Node D we find that our camera position is located in its frontspace, so we must render its back child first before we render the data stored at Node D. As Node D has no back child, we skip that step and just

render the polygon data stored there. This renders our first polygon so far (polygon D) as shown in Figure 16.34.

After rendering the data stored at Node D, we then step into its front child (E). Here we find a terminal node with no front or back children, so we simply render the polygon stored there, making polygon E the second polygon we render (see Figure 16.35).

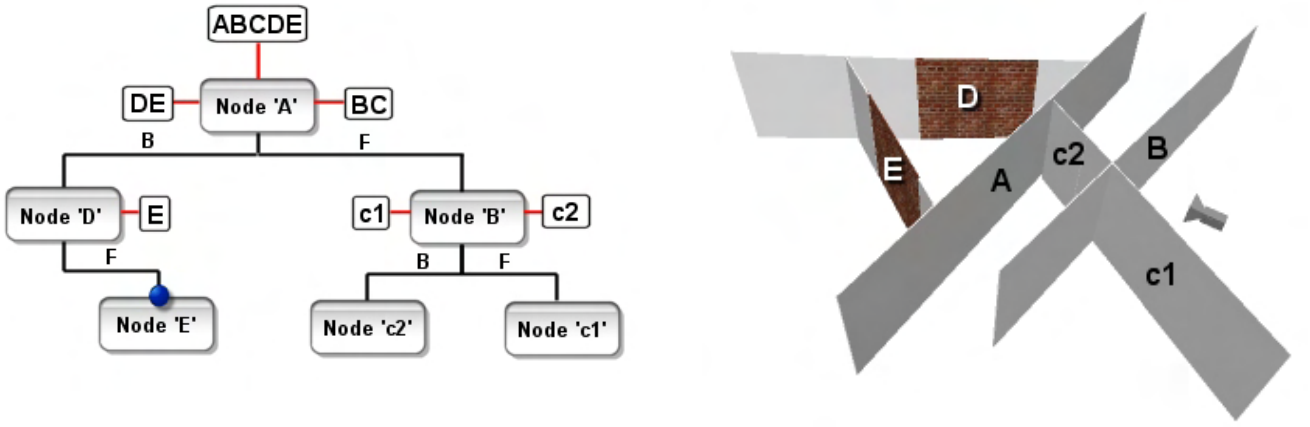


Figure 16.35

After rendering the polygon at Node E we pop back up to Node D and find that we have performed all the tasks necessary to process that node (having rendered its data and traversed its child list), so we return from that node and find ourselves all the way back up at the root node.

At the root node we have currently performed only one of the three tasks we must perform. As the camera was found to be in the node's frontspace we had to traverse into the back child first; we have now done that. The next thing we must do prior to stepping into the front list is render the polygon data stored at this node. This makes polygon A the third polygon to be rendered (see Figure 16.36). As we can see, so far, our scene is rendering in a perfect back to front order with respect to the camera position.

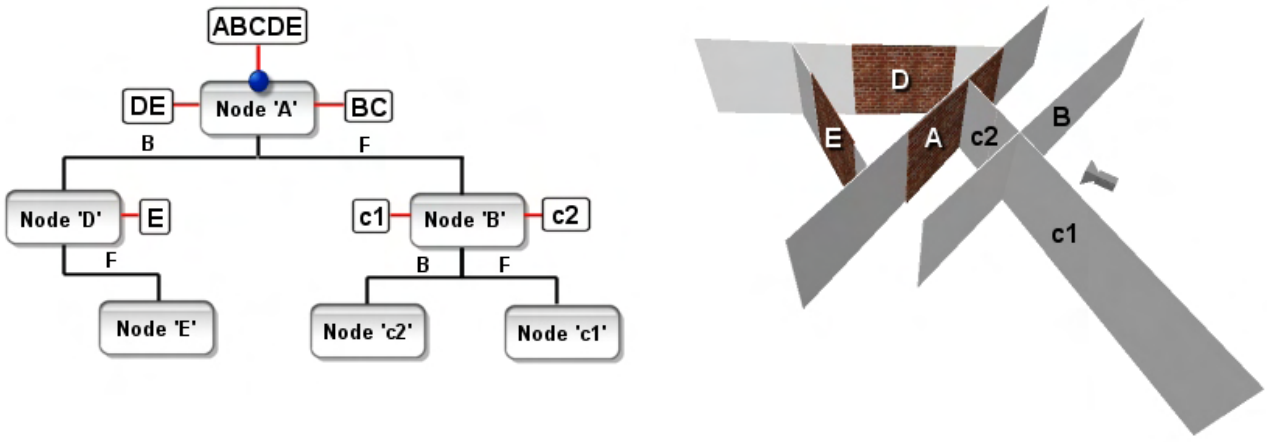


Figure 16.36

At this point we have rendered the data at Node A and should finally step into its front child where we arrive at Node B (see Figure 16.37).

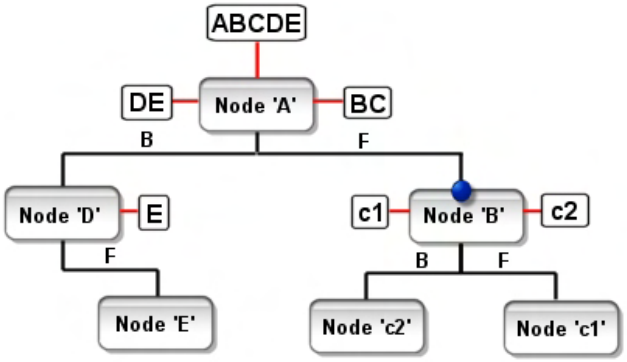


Figure 16.37

At Node B we classify the camera position against its plane and find that it is located in its frontspace. So we must step into its back child first, render the data stored at Node B, and then step into its front child. We step down into the back child of Node B first, where we arrive at Node c2. Node c2 is a terminal node, so there are no children to step into. As we now know, when this is the case we simply render the polygon data stored at the node (polygon c2 in this case).

As Figure 16.38 shows, polygon c2 becomes the fourth polygon we render, and we are still maintaining a perfect back to front draw order. This polygon did not exist in the original dataset sent into the tree; it was created from a larger polygon that originally spanned Node B. We can see now however, that polygon c2 can be perfectly described as being behind Node B.

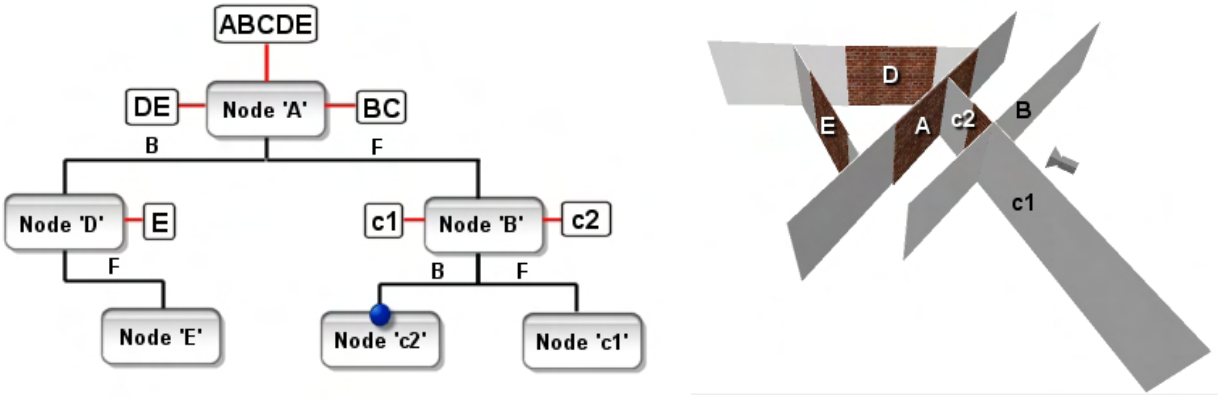


Figure 16.38

With the polygon data at node c2 rendered, we return and find ourselves back at Node B having processed its back child. Now we render the polygon data stored at Node B, which in this example is polygon B. Polygon B becomes the fifth polygon we render (see Figure 16.39).

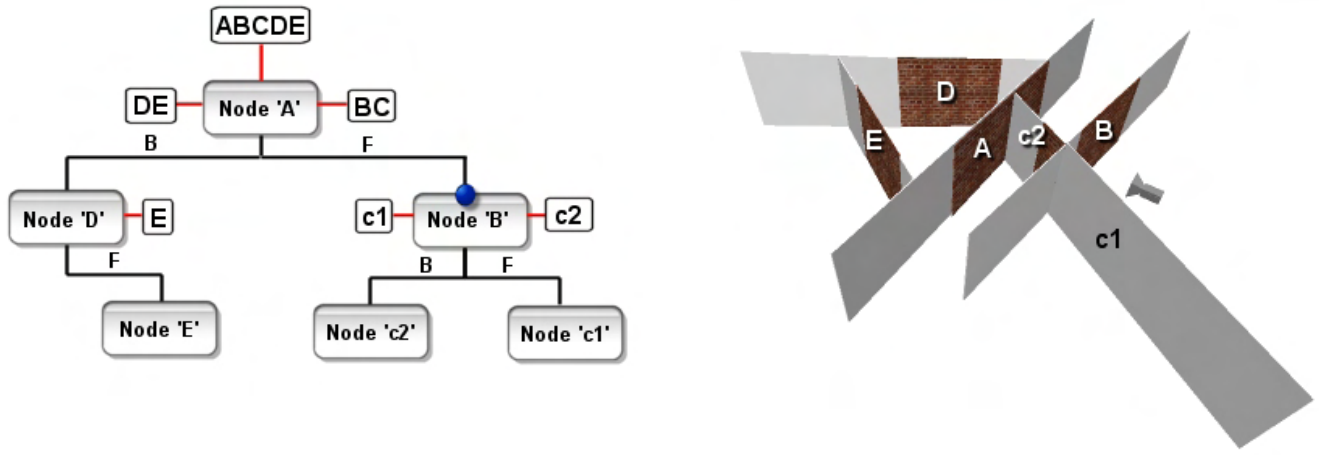


Figure 16.39

Having processed the back child of Node B and having rendered the data contained at Node B, our final task in the processing of Node B is to step into its front child where we arrive at Node c1. Node c1 is a terminal node, so we have to do nothing other than render the polygon data that is stored there. In this example that is polygon c1 (a child split of the original polygon C). This makes c1 the sixth and final polygon to be rendered, as shown in Figure 16.40.

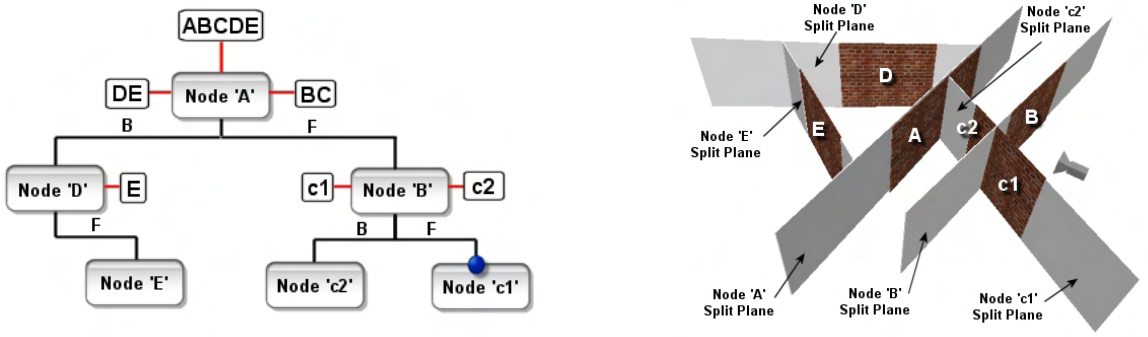


Figure 16.40

So by clipping the polygons to the node planes, we can generate a tree from an arbitrary polygon soup that can always be traversed in a perfect back to front order. This allows us to render polygons correctly such that nearer polygons will occlude more distant ones, even when we have no depth buffer support. While it is very unlikely given modern hardware that we would ever want to render our entire scene's geometry database using this technique, it certainly gives us a good solution for rendering our alpha polygons. This approach will ensure that blending always occurs in the correct order, even if multiple alpha polygons are layered on top of each other from the camera's perspective.

Since we will be building our tree such that on plane polygons that share the same normal direction as the split plane are stored at that node, the logic executed at each node during the rendering traversal is shown below:

- Enter node N
- Classify camera position again node plane
    - In Front or On Plane
        - Step into back child.
        - Render all polygons stored at node N
        - Step into front child
    - Behind or On Plane
        - Step into front child
        - Step into back child
- Return

Your attention is once again directed to the fact that if the camera is behind the current node, then the polygons stored at that node cannot be seen and are not rendered. If the camera position is in front of the plane, or situated on the plane, we render the polygon data.

One topic we have been deliberately steering clear from so far is the selection of a node's polygon as a split plane during the building process. While we can choose any polygon from the list passed into the node as the node's split plane and still get a BSP tree that works perfectly, there are some choices that are better than others when selecting a node's plane from a list of candidate polygons.

16.2.6 Choosing a Split Plane

We know already that our BSP compiler will recursively call itself with sets of polygons and that during iteration it will choose a polygon out of that list to become the splitter for the current node. All of the other polygons are then assigned to front or back lists, which themselves will be split, and so on until every polygon has eventually been chosen as a splitter. The last polygon left in a list will not actually split anything and will be assigned to the terminal node. The question is, during node construction how do we decide which polygon in the list should be used to create the split plane for that node? Is there some set of properties or some heuristic that we should consider to determine a preference?

While we could randomly pick a splitter each time from the list, there are actually some important distinctions that can be made between the candidates. That is, we can categorize polygons as either potentially good splitters or bad ones depending on certain criteria.

Every time we select a new polygon as a node's plane to divide a given subspace, any polygons in the list straddling the split plane (i.e., the splitter polygon's plane) will have to be split into two pieces. If you selected a splitter that intersected every other polygon in the list, all of them would have to be split into two pieces in order to be assigned to the front and back lists of the node. So on your first call to your compiler function you have just *doubled* the polygon count. This is definitely not a desirable outcome. Obviously the process continues as you traverse the new front and back lists and the polygon count (and therefore the node count) might quickly grow to a dangerous level with respect to

maintaining real-time rendering performance. It will also slow down the tree compilation time which, while not as onerous since it is an offline process, still impacts your development schedule.

While this example is a bit extreme, the point is that there are going to be some polygons that do not make great candidates as splitters early on in the tree creation process because they cause too many splits in the remaining dataset. So to build a more optimal BSP tree, we really want to consider candidate polygons that cause the fewest number of splits to occur in the remaining dataset.

Unfortunately, there is no way to build every potential BSP tree that would result from every potential splitter being accounted for at any given level in the tree. The number of permutations we would be talking about is a factorial based on the number of polygons (N!), so this approach is completely out of the question, even on a modern supercomputer. In case you were curious, or are unfamiliar with factorials, a tiny 50 polygon level would have $50! \approx 3x10^{64}$ possible BSP trees that would have to be built and examined for splitter preference.

So we can obviously forget about the possibility of building the perfect BSP tree, and instead settle for a very good one. One solution which gives good results is the following:

Each time we create a new node during the build process we will loop through each polygon in the list and process it. By 'processing' it, we mean that we will take its plane and test every other polygon in the list against it and record the number of polygons in the list that span that plane and would need to be split *if* the current candidate was used as the split plane. Whenever a polygon is processed which causes fewer splits than the previous minimum, we record the polygon and the split count. After every polygon in the list has been tested for a given node, we will have found the polygon in the list that will cause the fewest splits in the remaining dataset at that node and therefore is a great choice for the node's split plane. The process of polygon selection for a node's split plane is shown below. We might imagine that this is wrapped in a function called SelectBestSplitter which can be called by the node to select a splitter from its passed polygon list.

- Best Score = Infinitely High
- For each polygon in list A
    - Score = Splits = 0;
    - For each polygon in list B
        - Polygon B[*i*]->ClassifyPlane(Polygon A->Plane)
        - If (Spanning) Splits++
    - End for loop B
    - Score = Splits
    - if (Score < BestScore)
        - BestScore = Score;
        - pSelectedFace = Polygon A

- End for loop A
- Return pSelectedFace

As you can see, we first set the Best Score variable to a very high number. We will use this variable to search for the polygon that gets the lowest score (i.e., fewest splits) when tested as a split plane. We then loop through the polygons. The outer loop tracks the polygon being used as the split plane candidate. Inside this loop we set the current split count and score for this polygon to zero as we have not tested it against the other polygons yet. We then loop through every polygon in the list and classify it against the plane of polygon A. If polygon B spans polygon A then we know that choosing A as the splitter will cause this polygon (B) to be split, so we increase the split count. At the end of the inner loop we have tested every polygon (B) against the current split plane candidate (A) and have stored (in the variable Score) the total number of polygons that will get split immediately if this polygon is chosen as the split plane for the current node being constructed. If this score is lower than the best score we have recorded so far (for other polygons in the A loop) we record the score and the polygon which generated it. At this point we have stored the polygon which has generated the lowest number of splits so far. Once the outer loop has completed, every polygon in the list will have been tested as a split plane candidate and pSelectedFace will store the polygon that the current node should use as the splitter.

Although at first this might sound like the perfect tree compilation algorithm, keep in mind that choosing a splitter that causes very few splits at one level (node) of the tree could actually cause more splits further down the tree than might have been the case had we chosen a splitter which had initially split more polygons higher up in the tree. While we are content to work with this less than perfect approach to splitter selection, there is one other concept which we can factor into our choice of splitter, which is not new to us.

Although choosing a polygon that creates the fewest splits is arguably the most important criteria in splitter selection, there is also tree balance to consider. A perfectly balanced BSP tree (which is almost impossible to create without excessive splitting) is one with exactly the same number of nodes in both the back and front branches of the root (i.e., an even distribution of nodes). We discussed in the previous lessons the benefit of having a balanced tree and how it helps to achieve shallower trees and maintain consistent traversal times. Sometimes frame rate consistency is more important than pure speed. For example it is better to have a game engine run at 30fps consistently throughout the level than have it run at 90fps in some places and drop to 7fps in others. This is where tree balance becomes a factor.

Unfortunately, the downside to balancing a BSP tree is often the creation of more splits in the list. So we are going to have to find some middle ground here and adjust our split plane selection logic to also factor in the tree imbalance that will be introduced by choosing a particular polygon. As our split plane selection logic essentially has to classify every polygon against the candidate plane to record the number of splits, it is a small step to also record the number of polygons that are found to be completely in front or completely behind the candidate plane. If a candidate plane has the same number of polygons in its front list as in its back list, it means that the node splits its space in a perfectly balanced manner. The larger the disparity between the front and back face counts, the more imbalanced the node (and most likely the entire tree) will be. Therefore, we might imagine that the score for a given candidate polygon

would be the sum of the number of splits it creates and the absolute difference between the number of polygons found to be contained in both its halfspaces. This would make our polygon score calculation:

Score = abs(frontfaces - backfaces) + splits

So we will subtract the number of back faces from the number of front faces and take the absolute value. We then add to that the number of splits, giving us a score for the polygon that will be at its lowest when the polygon creates a perfectly balanced partition (frontfaces=backfaces) and causes no splits.

We are almost there, except that at the moment we are probably not giving enough influence to the number of splits. In most cases, reducing the number of splits will be our primary goal, so we will need a way of letting the selection process know that we would like to place more weight on the split count than on the imbalance score. Therefore, our SelectBestSplitter function will also take a split heuristic (a weight value) parameter which is multiplied by the split count to create the score for each candidate plane:

Score = abs(frontfaces - backfaces) + (splits * SplitHeuristic)

As you can see we now multiply the split count by the passed split heuristic so that we can give more or less priority to reducing splits by passing a higher or lower split heuristic value respectively.

Our new version of the SelectBestSplitter function will loop through the list of polygons passed in, choose a different candidate split plane each time and then test it against the rest of the polygons in the list. Each time a splitter has been considered, we will end up with the number of splits it caused and the number of front and back polygons that would end up in the front and back lists of the respective node. We can then calculate our score using the above formula. If the score is lower than any previous score we encountered during this call then we will keep track of this polygon. When the function ends, it will return a pointer to the polygon in the list with the lowest score. This polygon can then be used by the node to create its split plane.

- Parameter = “Split Heuristic” (single weight value)
- Best Score = Infinitely High
- For each polygon in list ‘A’
  - Score = Splits = Back Faces = Front Faces = 0;
  - For each polygon in list ‘B’
    - Polygon ‘B’->ClassifyPlane(Polygon A->Plane)
    - In Front :
      - Front Faces ++
    - Behind :
      - Back Faces ++
    - Spanning :
      - Splits++
  - End for loop ‘B’

    - Score = abs(Front Faces + Back Faces) + (Splits*Split Heuristic)

    - if (Score < BestScore)
        - BestScore = Score;
        - pSelectedFace = Polygon A;

- End for loop A

- Return pSelected

The logic to the code shown above is contained inside the CBSPNodeTree::SelectBestSplitter method in Lab Project 16.1. We will cover the actual source code to the BSP compiler to Lab Project 16.1 in the accompanying workbook.

16.2.7 BSP Node Trees Conclusion

We have now completed our introduction to the standard BSP node tree, using polygon-aligned planes to carve the geometry into pieces that can be rendered in a perfect back to front order. Although the node tree is the simplest of the BSP tree incarnations, you will be pleased to know that the leaf tree we develop in the next section is essentially exactly the same tree. The only real difference is some extra logic introduced during the build phase to collect convex clumps of polygons at the terminal nodes of the tree.

Covering the BSP node tree first was a good way to ease ourselves into the subject matter that we will need to explore both in this lesson and in the following lesson. However, let us not write off the BSP node compiler as simply an academic exercise with no real world application. Indeed, as we have discovered, even on today's latest cutting edge hardware, alpha polygons must be rendered in a back to front order to maintain proper blending. The BSP node tree is now going to be our tool of choice for handling alpha polygons. There are many other areas where BSP node trees can be used, such as in the realm of texture consolidation (efficiently packing multiple images onto a single texture surface). Also keep in mind that many applications it has in the area of geometry repair and constructive solid geometry.

Before moving on to the next BSP tree type, now would be a good time to open up the source code for Lab Project 16.1 and examine (along with the accompanying workbook) the node based BSP tree compiler.

16.3 Introducing BSP Leaf Trees

In this section we will examine a slight variation of the BSP tree building technique which will allow us to introduce leaves into our BSP tree. The BSP leaf tree is actually a lot more like the trees we have built in previous lessons because the polygons are not stored at the individual nodes but are passed down the tree during the build process and collected at the terminal nodes (i.e., the leaf nodes) of the tree. This is much more akin to the strategy used to build the quad-tree, oct-tree and kD-tree in the previous lessons. Although it may seem like we would now have to examine another form of tree, the good news is that this is not the case. Whether we build a leaf tree or a node tree, the tree itself is constructed in exactly the same way with respect to how the node planes are selected and the way that the space of the scene is subdivided. In fact, one might say that our node tree already has leaves; we are just not using them for anything yet. Closer examination of the BSP node tree that we constructed in the previous section will highlight the fact that although we decided to store the polygons in the nodes of the tree, the planes of the BSP tree still carve the scene into convex areas which can be navigated to by traversing the tree to the terminal nodes. Unlike the other leaf trees we have constructed however (quad-tree, oct-tree and kD-tree), the leaves of the BSP are not axis aligned bounding boxes, but are arbitrary convex regions bounded by the node planes which intersect to form that region. Yet just like any of our previous tree types, we can send objects or polygons down the tree and assign them to the terminal nodes in which they are eventually found to reside. We will see that these terminal nodes represent a convex region of the scene and what is considered to be a ‘leaf’.

This all sounds a little abstract at the moment, so let us take the node tree that we looked at in the previous section and examine what would be involved in turning it into a leaf tree. Figure 16.41 shows the BSP node tree we constructed in the previous section.

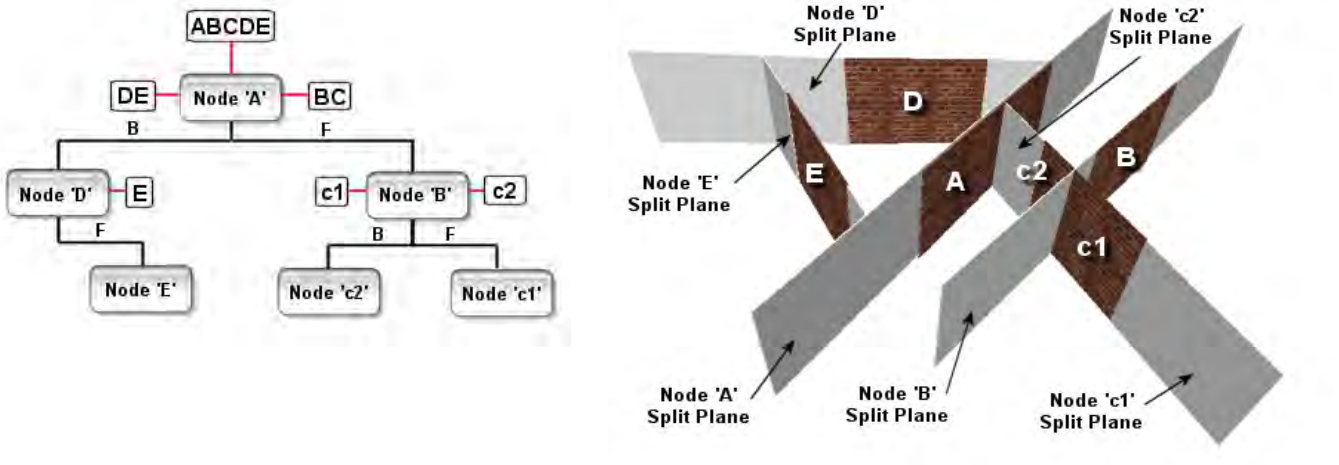


Figure 16.41

The node tree is referred to by its name simply because the polygons are stored at the nodes. Recall that this was done to solve a very specific problem related to alpha sorting. By realizing that the planes of the tree could be traversed in a back to front order and that the planes themselves were created from the polygons, every polygon in the scene will lay on one of these planes. It is logical to assume then, that we

can render the polygons in a perfect back to front order by storing them at the nodes they helped create and render them when those nodes are visited during an ordered walk of the tree. This is obviously a very useful strategy to use when perfect back to front order rendering is required. However, it would be extremely inefficient to also store our non-alpha polygons in this way due to the fact that we essentially process one polygon at a time. It would be much nicer to have a BSP tree at our disposal which works much like the quad-tree for example, where the polygons are collected at the leaf nodes and can be rendered in a batch when the leaf is found to be visible. Furthermore, although we already have trees at our disposal that allow us to do just this, you will see later that the leaf BSP tree will become one of the most important technologies we will implement in this course as it will allow us to construct potential visibility sets, do hidden surface removal, and perform a host of other really useful and essential tasks. Therefore, while the BSP leaf tree may seem like just another leaf tree (which it is), we will see later that because the polygon data is used to construct the node split planes, the tree is built with very important information about which areas of the scene are considered to be empty space and which areas are considered to be solid space. This is the only spatial tree we have covered so far that provides us with this information. This is precisely the information we will need in the following lesson to build a potential visibility set compiler for our scene data and accelerate rendering by an order of magnitude.

To understand where the leaves are located in a BSP tree and what they represent we will first highlight the similarities between the BSP tree and the other tree types. For example, we know that we can build an empty quad-tree by simply dividing empty space and assigning no polygon data to it. The result is a tree of empty bounding boxes. Such a technique can be useful if a game uses purely dynamic objects and would like to benefit from hierarchical spatial partitioning. The quad-tree starts with a large root node bounding box and subdivides that box down to a certain level. When the tree has finally been constructed, there are no polygons stored in the tree, but the leaves are still there -- they are just empty. We still have the ability to traverse the tree and find visible leaves and we will still have the ability to send volumes down the tree and assign them to the leaf nodes in which they are found to reside. The same is essentially true with our BSP node tree. That is, we have subdivided the scene into convex areas using a series of planes and as such, navigating to the terminal nodes allows us to describe ourselves as being in one of those areas. In the case of the node tree, we are just choosing to store nothing at the terminal nodes that represent those areas. Therefore, although the node tree did not store any polygon data in the leaves of the tree, the leaves still exist and are implied by the nature of spatial partitioning.

Let us imagine that we decide to build the same node tree illustrated in Figure 16.41 but decided not to store the polygons at the nodes. We will simply discard them after they have been used to create a node plane (along with any co-planar/same facing polygons). That is, after a polygon has been selected as a splitter and used to create a node plane, that polygon, along with any others that are co-planar and same facing, are deleted from the list and are never considered again during tree creation. The result would be an empty BSP tree just like that empty quad-tree we just discussed. The polygon data passed into the tree in this example was simply used to determine which split plane to use at each node. In fact, one can imagine how the BSP compiler could even be modified **not** to take a list of polygons but instead a list of planes in such a scenario. The resulting tree would be the same (see Figure 16.42). The only difference now is that we have not stored the polygons at the nodes. We have an empty BSP tree just like that empty quad-tree we discussed above, just carving up space using a different set of criteria.

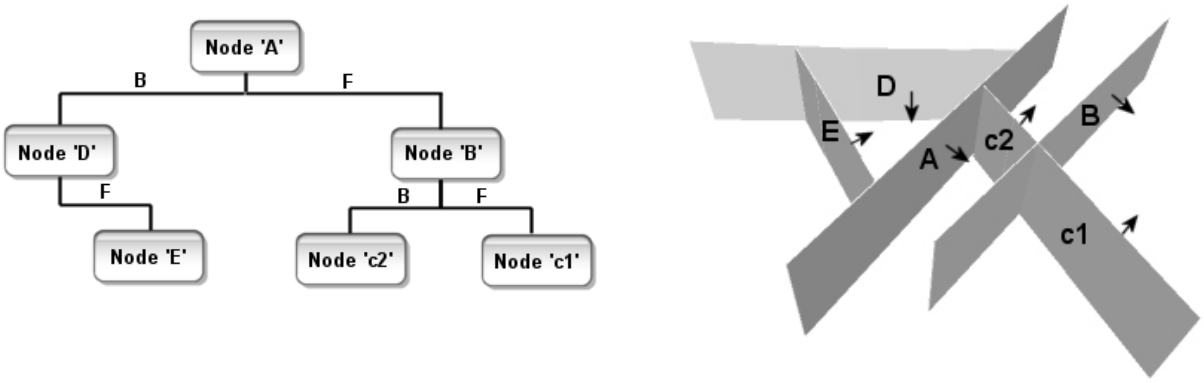


Figure 16.42

In Figure 16.42 the plane normals are shown as the black arrows in the rightmost image so that we can see the direction in which the planes are facing. When looking at the rightmost image and remembering that planes are infinite we can see that the planes divide the scene up into a series of convex areas.

> **Note:** Although planes are technically mathematically infinite, when drawing the planes of a BSP tree, the planes only carve up to the parent node space. This is because the plane is only used to carve up the designated halfspace of the parent node. Node D for example is assumed to carry on infinitely to the right of the image but stops at node A because it was selected to only partition node A's back space.

We will examine why using the polygon planes as the node planes of the tree divides the scene into a series of convex areas a little later, but for now just know that this is the case. For example, we can see that planes A, D, and E bound a triangular region of the scene. This region exists behind node A, in front of node D and in front of node E. Therefore, we can see from the tree diagram on the left hand side of this image, that if an object is found to be in front of node E during a tree traversal, it must be in this area and therefore, this area is the front leaf of node E. What structure we use to represent this area is not important for the moment (we might imagine that we use a structure that can contain a list of polygons and dynamic objects that have been passed through the tree and found to exist in that area). Furthermore, we can also see that if an object was found to be in front of node c2, it is obviously in the area of space bounded by planes A, c2, and B labeled 'leaf 3' in Figure 16.43 since node planes A and B were also navigated to reach node c2.

The conclusion we can draw from this discussion is that every terminal node in our tree is a plane which has a region down both its front and back sides. That is, each terminal node has a back leaf and a front leaf which represents one of the areas making up the scene. In fact, a leaf exists wherever a node has no front or back child. In this instance, we are referring to a leaf as a region of the scene that is bounded by the planes that have been traversed to reach that area of the tree. In Figure 16.43 we have color coded

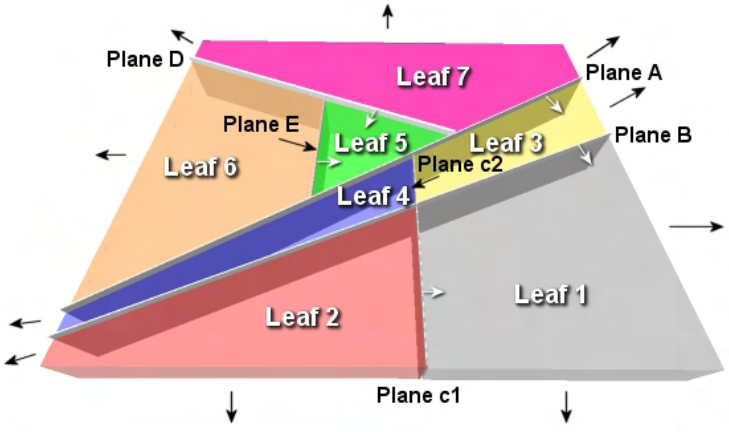
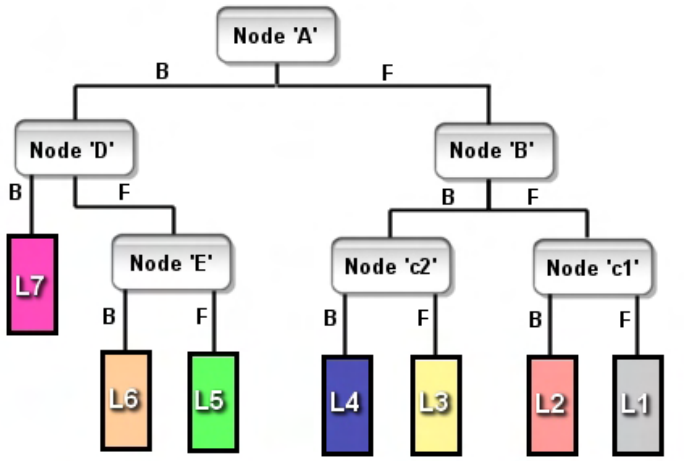


Figure 16.43

the convex areas of the scene and have also illustrated in the accompanying tree diagram where these areas exist.

Looking at the tree diagram for Figure 16.43 we can see that we have added leaves to the tree where a node has no front and back child node. To be clear, these leaves (labeled L1 to L7) are all empty and are labeled here to illustrate which area of the scene is reached by stepping down the front or back of a childless node. The boxes in the tree diagram might represent a leaf structure where polygons and dynamic objects are stored. The important point being made here is simply that whenever we step down the side of a node and it has no child node, we are in a leaf. This is a convex area described and bounded by the planes above it in the tree.



> **Note:** The leaves L1 through L7 could be represented as simple structures used to contain any objects or polygons that get assigned to that area. The important point is that we recognize this is really no different in principle from the node tree. The tree is exactly the same and the regions represented by these leaves always existed, the only difference now is that we are assuming the use of a leaf structure to catch and store any objects that get passed down the tree and eventually end up getting passed down the front or back of a node for which no child node exists.

Study the diagram and try out some test traversals of your own to see if you understand why the leaves are located in the tree in the locations that they are. Do not worry if you are still finding this a little confusing, we will step through some examples.

Figure 16.43 shows us that the BSP tree carves the scene up into regions that we call leaves and as such, is not unlike the quad-tree, oct-tree and kD-trees that we built in previous lessons. There are two big differences: First, a leaf is no longer an axis aligned bounding box, but is defined by the planes that bound the region (the planes above it in the tree). Second, not all leaves are fully surrounded by planes and as such, are assumed to carry in infinitely in the unbounded direction (this is only true for exterior leaves). In our simple example scene, all leaves except leaf 5 are exterior leaves, as they exist around the exterior of the scene. For example, we can determine the exact volume of leaf 5 since it is fully bounded by planes A, D and E. These are the parent planes of the leaf that need to be traversed in order to make it into that region. However, leaf 6 which is located behind node E is only bounded by planes on three sides and is therefore assumed to have infinite volume to the right of the image. We can also see that leaf 1 would also continue infinitely down and to the right of the image as the leaf is only bounded by planes on its left and top sides.

Let us now use some example locations which we will send through the tree and test that our theory is correct. In Figure 16.44 we have placed a red sphere in a region of the scene that visually places it in leaf 3. That is, it is located behind plane B and in front of planes A and c2. We might imagine that this red sphere represents the position of our camera in the scene and we would like to know in which leaf within the BSP tree it is currently located so that we can render any dynamic objects that have also been assigned to that leaf (the structure attached to the front of node c2). Finding what leaf a given position vector is in is no different from the kD-tree case. We simply send it down the tree starting at the root node and classify it against each node plane we visit, sending it down the front or back of the node depending on the result. Figure 16.45 shows us sending the query position shown in Figure 16.44 through our BSP tree.

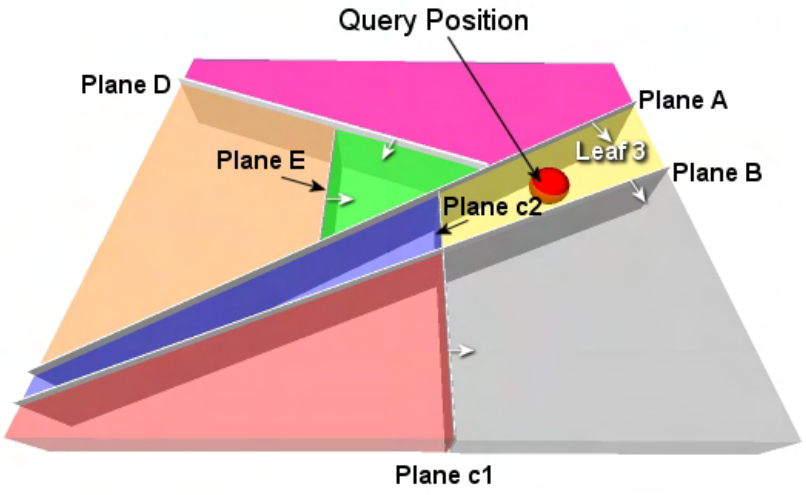


Figure 16.44

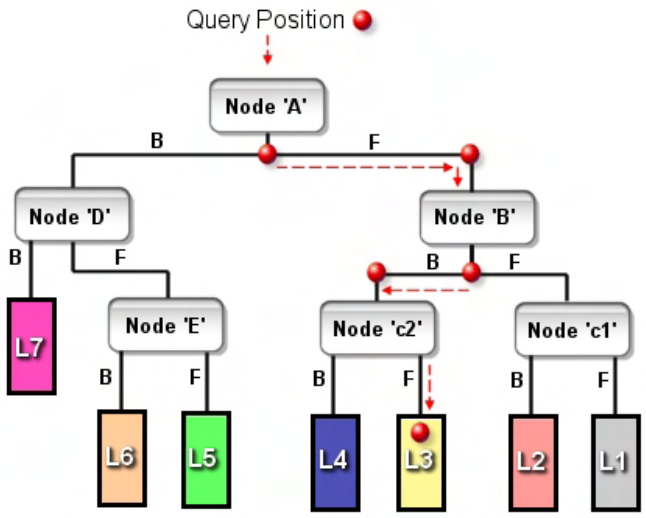
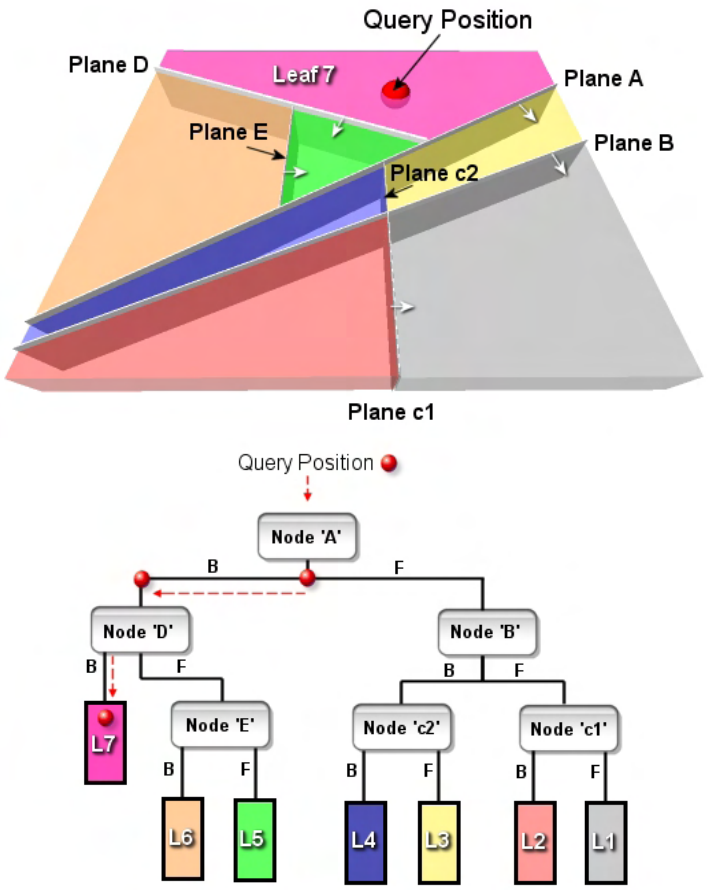


Figure 16.45

In Figure 16.45 we show the position in Figure 16.44 being passed through the tree. You should reference both of these images during this next examination of the traversal process.

At node A we classify the query position against the node and find that it is in front of plane A (as can be seen in Figure 16.44), therefore, it is passed down the front of A where it reaches node B. At node B the query position is found to be behind the plane stored there and is passed down the back of node B where it enters node c2. At node c2 the classification is performed once again and this time the query position is found to be located in the front halfspace of the terminal node c2 which places it in leaf 3, the region of space bounded by planes A, B, and c2. Leaf L3 might be a structure that contains all the polygon data that has been passed down the tree and clipped to the nodes and found to reside in c2's front halfspace. These polygons would also need to be rendered if leaf 3 was found to exist inside the view frustum.

In Figure 16.46 we see another example of traversing the BSP tree with a query position and further solidify our understanding of the regions of space represented by the leaves of the BSP tree.



The query position is shown in Figure 16.46 to exist in leaf 7. Looking at the topmost image we can see that this is a region of space that is bounded by planes A and D. It is located behind both of these planes. Thus, if we know that a given position is located behind planes A and plane D then it must be in the region of space we have labeled leaf 7. We can see that this is the case in the topmost image and furthermore, can see in the bottommost image that when we traverse the BSP tree with this query position, these are exactly the tests that are performed. The query position is fed in at the root node where it is found to be located behind node plane A. Because of this, it is passed down the back tree of A where it reaches node D. The query position is once again classified against node D where it is found to lay behind that node also. As node D has no back child we know that this is the end of the road and we have reached the convex area that we have labeled leaf 7. Finally, we will show one more example where the query position is located in leaf 5 accompanied by the tree diagram that shows the query position being passed down the tree. Eventually we will pop out in the front space of node E.

Figure 16.46

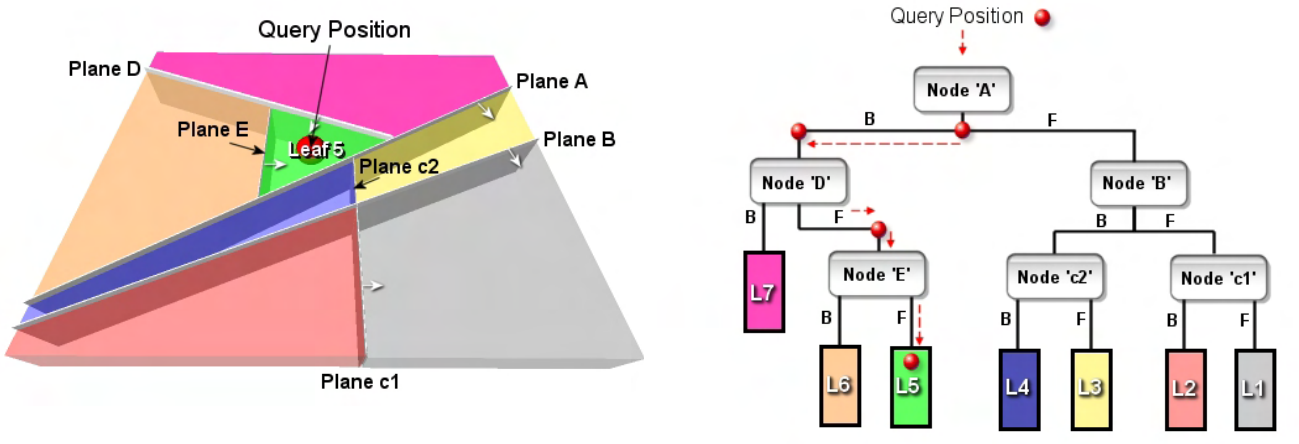


Figure 16.47

When we look at the rightmost image in Figure 16.47 we can see that if we were to describe this position in English we would say that it is in the region of space behind plane A and in front of planes E and D. However, notice that in the rightmost image this is exactly what we are doing when we traverse the BSP tree. We are testing the query position against these planes of the tree and eventually find that our query position is located in front of node E, in front of node D and behind node A.

16.3.1 Populating the BSP Leaf Tree

Now that we understand where the leaves of the BSP tree exist and what areas they represent, it is the next logical step to determine how we would compile the leaf tree and collect the polygon data at the leaves of the tree instead of storing them at the nodes. Before we look at how to modify the BSP build process to collect the polygons at the leaf nodes during the compile process, we will first perform the polygon population of the tree in a separate pass. This will give us the opportunity to become more familiar with the rules of passing polygons down the BSP tree before we merge this polygon passing and collection logic into the core build process. Therefore, in these next examples we will assume that the BSP tree was built as in our previous example and is an empty tree initially. That is, during tree construction, after a polygon was used as a splitter, it was simply deleted from the list and considered no further. As we saw in the previous section, this creates a tree of separating planes only. Once we have this tree we will then send the polygons that were used to create the tree into the root node one at a time and will track their progress as they descend through the tree into the leaf nodes in which they belong. This will demonstrate the polygon passing logic that we will eventually merge into the core build process.

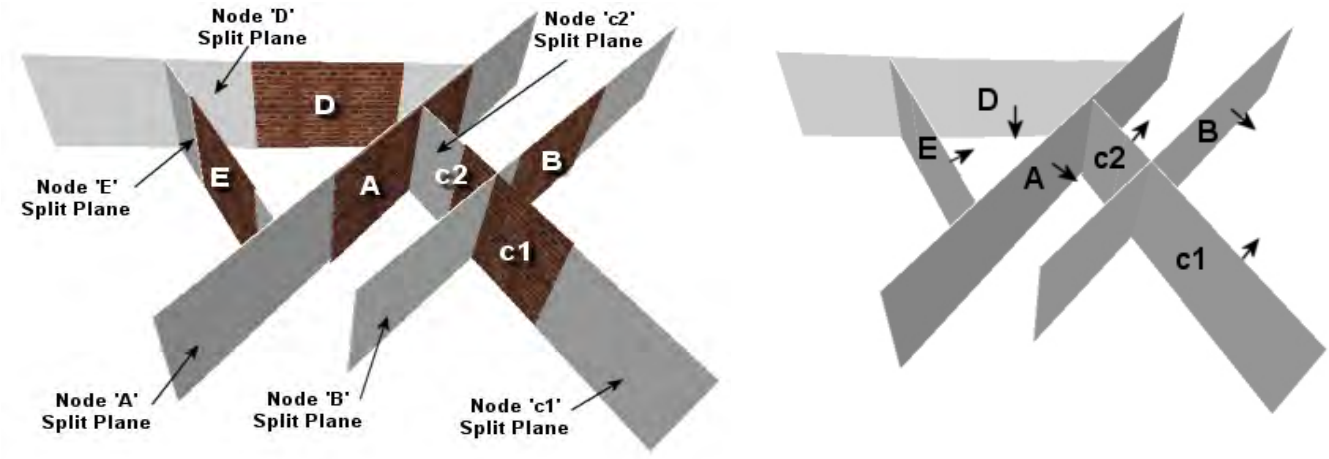


Figure 16.48

The leftmost image shows the polygons that were initially passed into the tree and also shows the node plane created by each. The rightmost image shows the BSP tree after compilation with no polygon data assigned to it. As you can see however, the polygon data was still used to determine the node planes to be used.

Assuming that our tree has been constructed as shown above, what we will now do in a second pass is loop through polygons A through E and send each one down the tree starting at the root node. At each

node, the polygon will be classified against the plane and sent down the front or back child depending on the classification result. If a polygon is found to be spanning a node plane then it will be split by that plane and the two child fragments dispatched down the front and back of that node respectively. If the polygon is found to lie on the plane and face in the same direction as the node plane it will be passed down the front of the node, if it is co-planar but faces in the opposite direction to the node plane it will be passed down the back of the node.

> **Note**: Later in this lesson we will learn why dealing with the co-planar classification case as described above is so important. With the leaf tree, we must pass co-planar same-facing polygons down the front of a node and co-planar opposite-facing polygons down the back of a node. Take this on face value for the time being as the reason for this will not become clear until we wish to exploit the solid/empty space determination properties of the tree. However, if we do not handle co-planar polygons in this exact way, we will lose the ability to query the solid/empty space properties of the geometry stored within the tree.

Figure 16.49 shows us that polygon A will be the first to be passed down the tree. Figure 16.50 shows the journey of this polygon through the tree where it eventually gets clipped into two fragments that exist in leaves 3 and 4 respectively. The first plane it is tested against is node A. As it is co-planar with this plane and pointing into the same frontspace, it is passed down the front to node B. It is found to lie behind node B so is passed down the back to node c2. Polygon A spans node c2 so is split into polygon fragments a1 and a2 and the original polygon A is deleted. Polygons a1 and a2 are passed down the front and back of node c2 respectively. The two fragments end up in leaves 3 and 4.

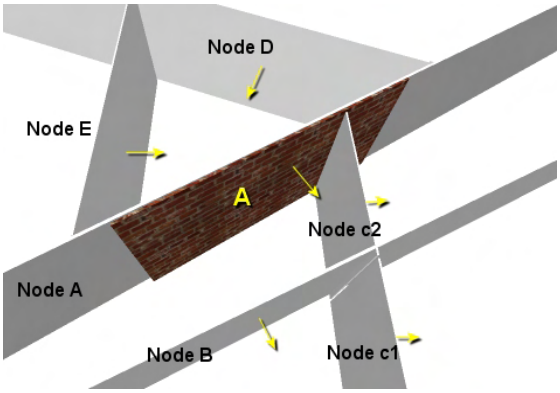
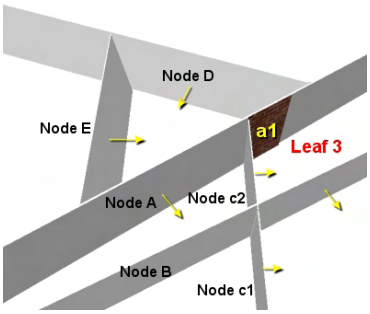
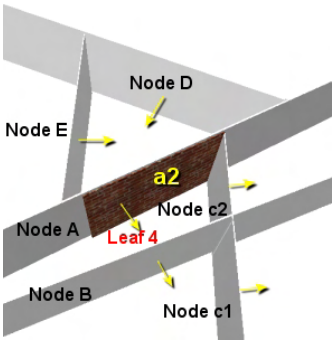


Figure 16.49





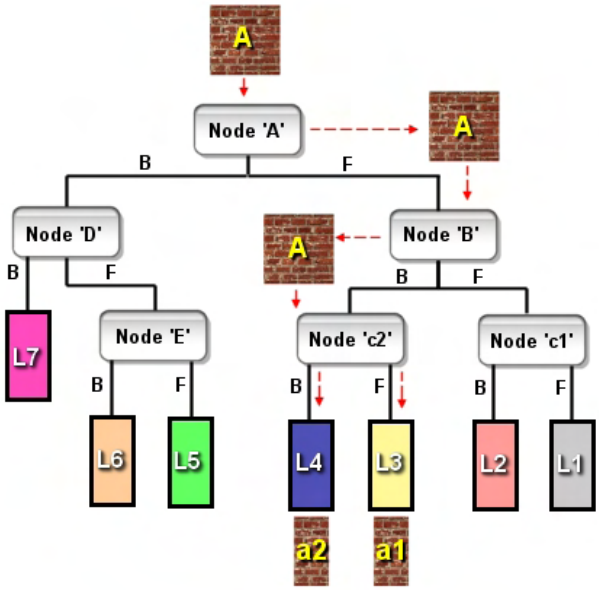


Figure 16.50

At this point then, we have two polygons stored in our tree, a1 and a2, which were both originally created from polygon A. These polygons will have been added to leaves L3 and L4. Next we send polygon B through the tree. Its journey is shown in Figure 16.51 where we see that it ends up in leaf 1.

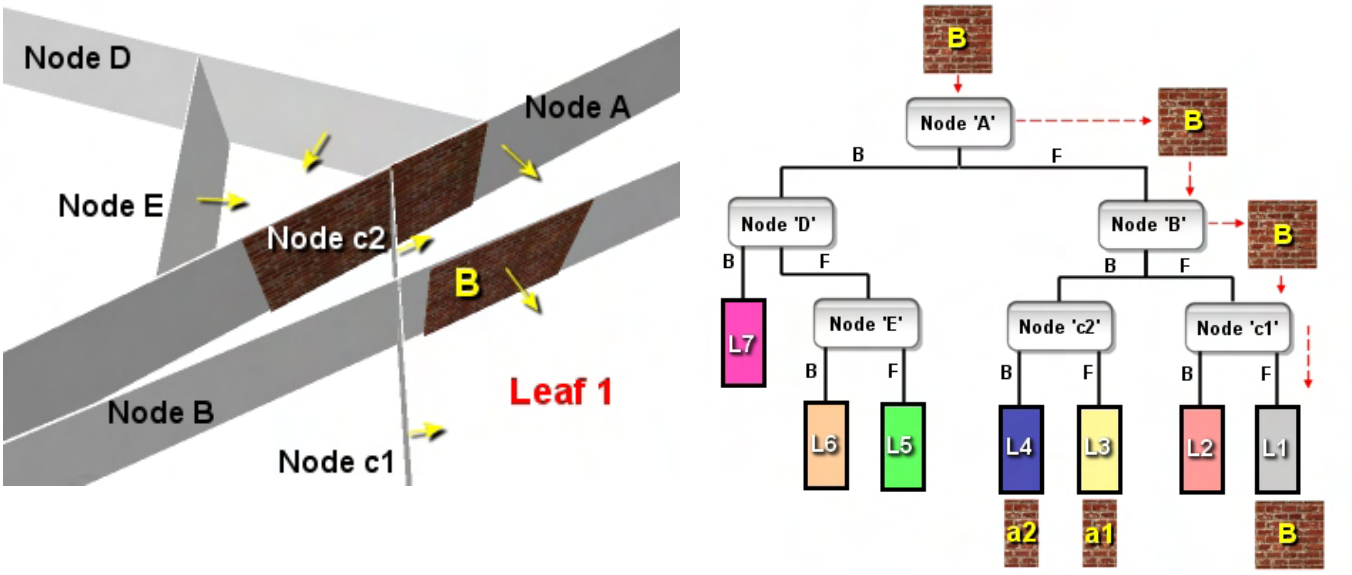


Figure 16.51

We would like to draw your attention to a very important trait of the BSP tree which we are already starting to see manifest itself, even with the two polygons that we have added to the tree thus far. In the quad-tree, the oct-tree and kD-tree, although the leaf nodes represented convex regions within the scene (in these instances the region was an axis aligned bounding box), the polygons were assigned to these regions as a soup. That is, the leaf nodes contained a soup of polygons which filled up the space of the leaf node to some extent. However, this is not the case with the BSP tree.

Because the polygons themselves are used to create the split planes we know for a fact that every polygon in the tree will lay on one of those node planes. As we also know that the node planes always form the boundary of one or more leaves, it stands to reason that the polygons assigned to a leaf will actually bound the region represented by that leaf and will not be contained in the middle of the leaf's space. We can clearly see that this is the case in all of our diagrams to date. In Figure 16.51 for example, we can see that although polygon B was assigned to leaf 1 (the leaf it faces into), it is located on the boundary plane of that leaf. This is a very important point to bear in mind as we go forward.

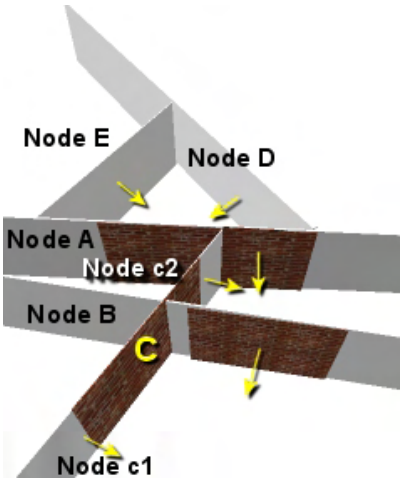


Figure 16.52

In the next example we will feed polygon C into the BSP tree. The original polygon C is shown in Figure 16.52 and we can already see that it will be split into two fragments and will therefore live in two leaves. Figure 16.53 shows the polygon's journey through the BSP tree.

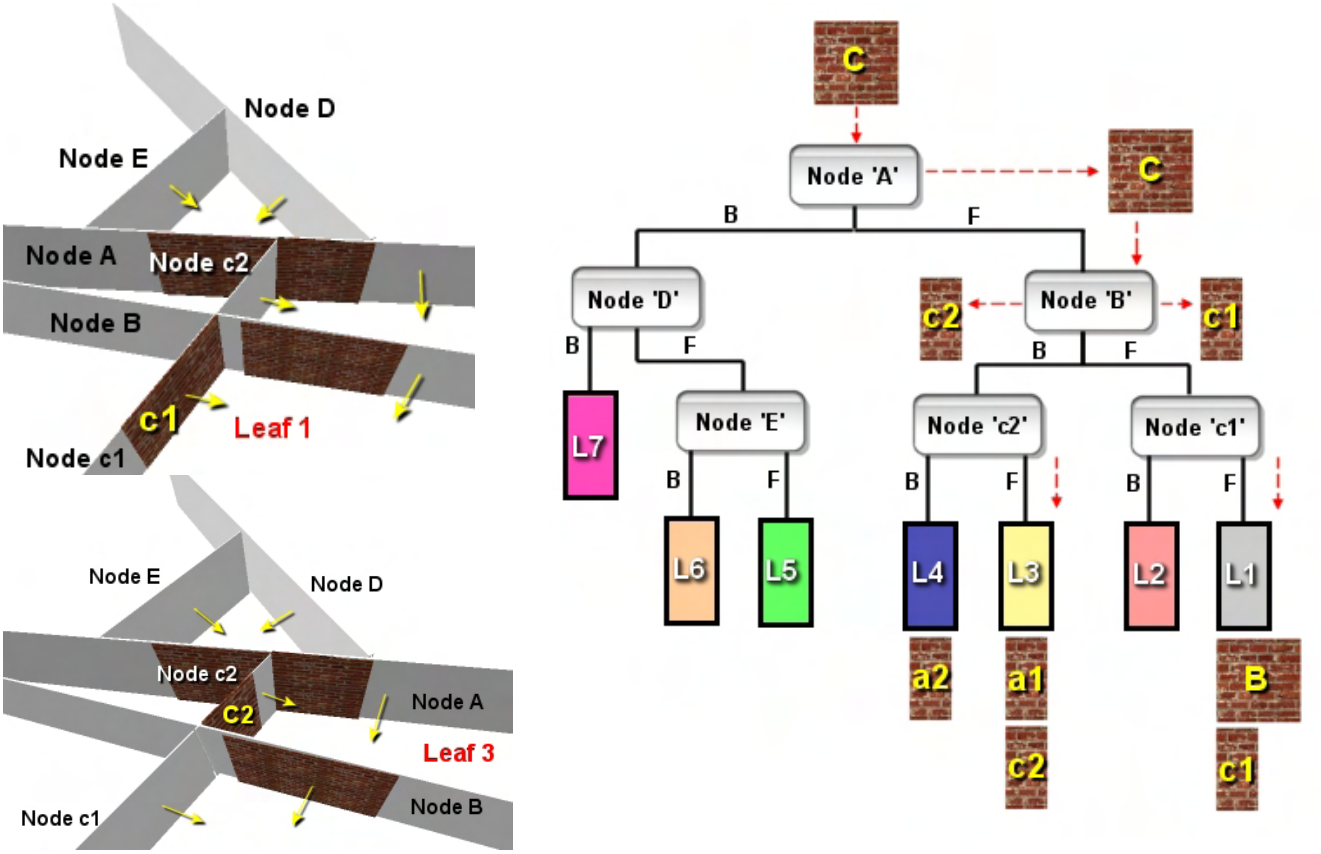


Figure 16.53

As the rightmost image in Figure 16.53 shows, polygon C is fed into the tree and is classified against the root node. Polygon C is found to be located entirely in the front space of node A so is dispatched down the front side to node B. At node B a similar classification takes place between the polygon and the node plane and this time polygon C is found to be spanning node B. This means polygon C will have to be split into two new fragments, c1 and c2 and the original polygon C will be deleted. Split fragment c1 is in the front halfspace of B so we will follow that fragment first.

As c1 is passed down the front of node B it arrives at node c1. It is obvious in this example that c1 is on plane with node c1 as this was the polygon that was originally selected to create the node's plane. Remember, we have built the tree and populated it in two different passes in this example. As polygon c1 is co-planer with node c1 and has a normal oriented into the node's front space it is passed down the front of node c1. However, node c1 is a terminal node so we know that to the front of this node must exist a leaf and in this example, that leaf is leaf 1. Polygon c1 is added to leaf 1's polygon list where it joins polygon B that was added to the leaf a moment ago. Polygon c1 has been fully processed and assigned to its leaf node so we unwind back up to node B again where we still have to process split fragment c2, which is located in its back space. Polygon c2 is passed down the back of B where it is found to be co-planar with node c2 and is therefore added to leaf 3 where it joins polygon a1.

With polygons A,B and C assigned to leaves in the tree, next we pass in polygon D whose original location is shown in the leftmost image in Figure 16.54. The rightmost image shows its route of classifications as it is passed down the back of A to node D. At node D it is found to be co-planar and

facing into the same frontspace and is therefore passed down the front where it arrives at node E. Polygon D is also located in node E's frontspace so is finally passed down the front and stored in leaf 5.

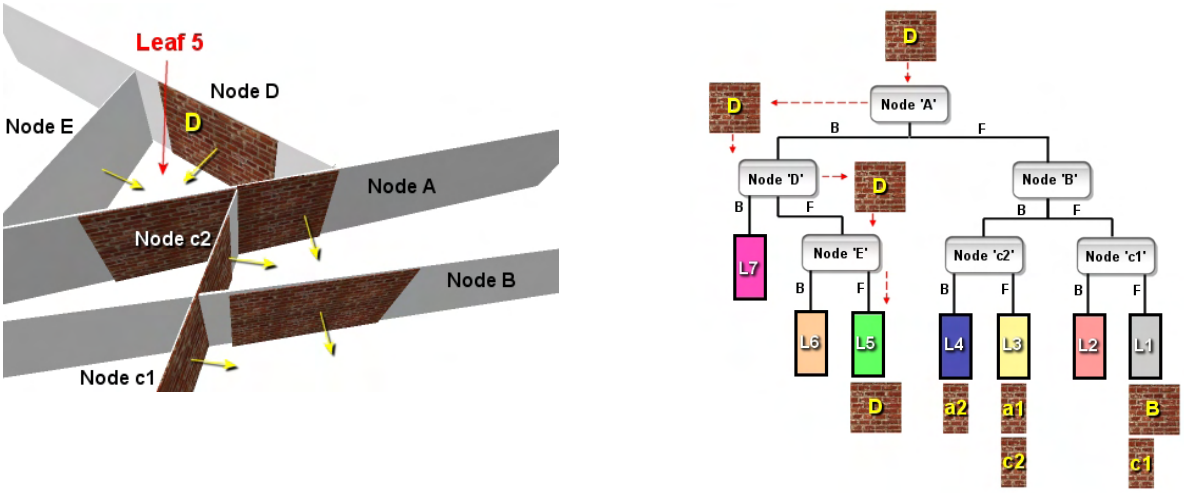


Figure 16.54

Finally, we have one polygon left to process, polygon E. It takes an identical route through the tree as polygon D (see Figure 16.55). It too ends up being assigned to leaf 5 where it joins polygon D.

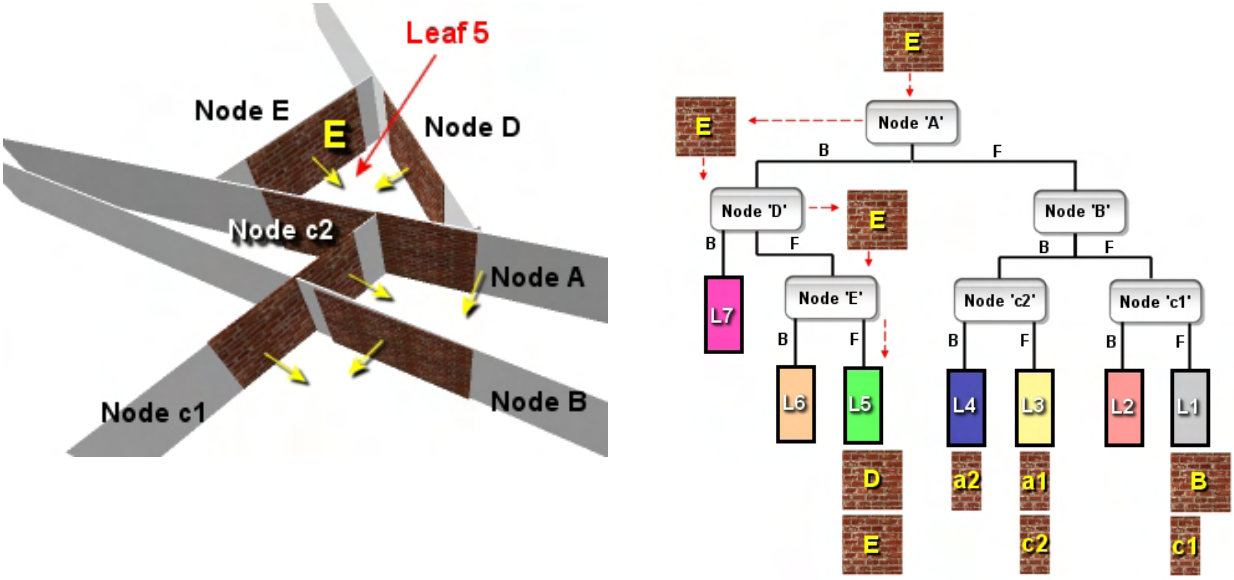


Figure 16.55

To clarify the process, we have performed the BSP compile in two different passes. Hopefully this has highlighted for you just how much the BSP tree is like any other tree we have developed so far. It has also shown us that the node tree also is the exact same tree as the leaf tree with the exception that we chose not to use the leaf information that was implied by the partitioning scheme. Our original node tree example now has been converted into a leaf tree and the final result is shown in Figure 16.56

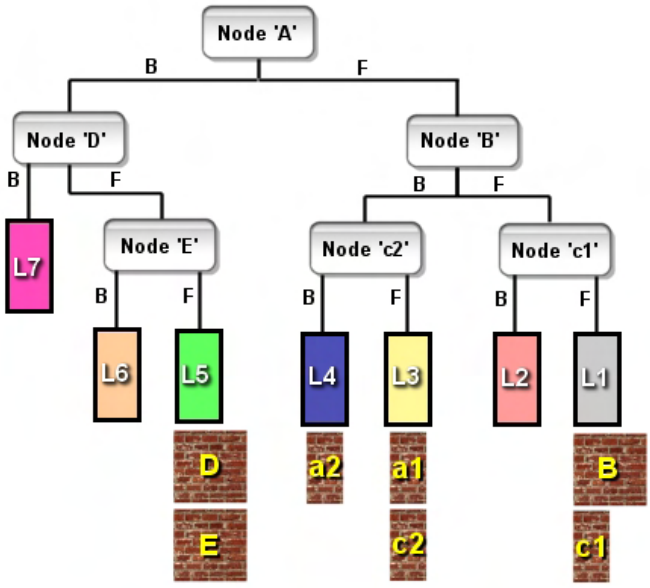


Figure 16.56

Notice that some leaves have had no polygon data assigned to them (leaves 2, 6 and 7). Also notice in all the previous diagrams that because of the way that we send a polygon that is co-planar and same facing down the front of that node, a polygon will always be assigned to the leaf that it is facing into. It might seem strange to think of the camera being in leaf 5 and not being able to see polygon A whose plane also bounds that region but polygon A would be back-facing and therefore would be back face culled by the pipeline. ***The only polygons assigned to a leaf are the ones that exist on one of the leaf's bounding planes and that have normals that face into that leaf.***

This would seem to cause a bit of problem. If the camera was located in leaf 5 for example, we would expect to see polygon A would we not? This is true, but this whole problem is caused by the fact that we have used as our example (quite deliberately) a completely infeasible data set. That is, we have developed data where it is possible for the player to see the backs of polygons, which is something you would never be allowed to see in a real scene. As we know, back face culling means that if we were to ever be allowed to walk around a polygon and view its back side, it would essentially disappear in front of our very eyes causing a complete breakdown in the solid integrity of the level. You will see shortly that when provided with proper legal geometry, this situation will never arise and any leaf which represents empty space (the space in which the player is allowed to be) will be bounded only by polygons that face into the space of that leaf.

In the above examples we built a tree of planes first and then pumped the polygons through the tree in a second pass to populate the leaf nodes. While this method certainly works it is a little redundant to perform a second pass when we already have the polygon data available during the first pass. Later, we will show some example code of how to merge these two processes together to create a function that will compile a BSP tree from an input polygon list and populate the leaf nodes with the polygon data as it is being constructed. While this might sound quite complicated, it really is not. When you think about it, this is exactly how our quad-tree, oct-tree and kD-tree were built in the previous lesson. There are some added complications for sure due to the fact that the polygon list is not only being passed down the tree and collected at the leaf nodes but is also being selected to construct node planes at each step, but they are trivial and can be resolved with a line or two of conditional logic. However, for the time being we will stick with the two pass approach which will help us more clearly demonstrate certain topics over the coming sections.

16.3.2 BSP Trees and Convex Areas

In this section we will briefly discuss how and why the BSP tree compiler always manages to break the most complex level into a series of simple convex areas. This will also highlight the BSP tree's ability to represent any complex mesh internally as a series of convex areas that can be easily queried for point/ray/etc. containment. This is important because we will often wish to determine whether a point is contained within a non-convex object, which is much more expensive than testing against a simple convex volume. It should be noted that you do not need to know this information in order to build a BSP tree but may find it an interesting read and helpful to really understanding the inner workings of the BSP compilation process.

Perhaps the best first example of how binary space partitioning using the polygon planes works to break the scene down into convex areas can be shown by examining the process of breaking a non-convex polygon into a convex one. All the technologies we have developed thus far are designed to work with convex polygons and therefore, this information may become useful to you if your scene has been designed in such a way that it contains non-convex polygons. In such an instance, you will have to isolate these polygons and break them down into a series of convex ones. This is not something you will usually ever have to do as most level editors and modeling applications will enforce the exportation of convex polygons and perform the convexity process for you. However, imagine if you were writing your own world editor like GILES™ and you allowed the user to edit a polygon at the vertex level. It is quite possible that the vertices of the polygon could be manipulated in such a way as to form a non-convex polygon. It is important that the editor break this up into convex polygons as our collision routines and point in polygon tests rely on convex polygons being used.

Figure 16.57 shows a non-convex polygon. A useful definition of a convex primitive is one that states that the infinite planes which form the boundary of an object must **never** intersect that same object's interior space. If we were to apply that same definition to a polygon, this would mean that if we were to iterate through each of the polygon edges and construct planes from them, at no point should the polygon itself be found to be spanning any of those edge planes if it is to be considered convex. In Figure 16.57 an edge is highlighted red which clearly shows that its infinite plane would cut through the polygon's interior.

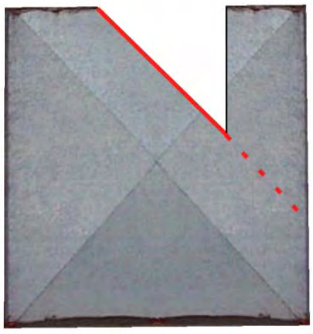


Figure 16.57

Using such a primitive would completely break our point in polygon intersection tests because we can no longer test whether a planar point is interior to a polygon simply by testing if the point is contained behind all of its edge planes (as we can in the convex case). It is quite possible for a point to be situated within the interior of the polygon shown in Figure 16.57 but still be located in front of one of its planes. Our point in polygon test would reject this point as soon as it was found to be in an edge plane's frontspace and would assume that the point is not interior to the polygon. You should be able to easily find a region in this polygon where a point could be located in front of the edge plane highlighted red in Figure 16.57.

Imagine that we wished to test if this polygon was convex or not. We could loop through each edge of the polygon and construct a plane from it (we learned how to do this in earlier lessons). For each edge

plane we would classify the points of the polygon against it. If we find an edge plane that has vertices in both of its halfspaces, we know this is a non-convex polygon and thus we split the polygon along that plane into two child polygons which are hopefully convex. Figure 16.58 illustrates that once the red edge plane in Figure 16.57 has been located, we could split the polygon by this plane thus creating two convex polygons which can be used in its place. The original non-convex polygon can then be discarded.

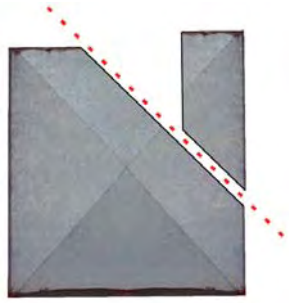


Figure 16.58

In this particular example our task would be complete because by finding this spanning plane and splitting the polygon, the result is two convex polygons which can be used instead of the original. We can now see that we can determine if a point is inside any of these two convex polygons simply by testing if a point is contained behind all of their edge planes (or in front of the edge planes depending on the directions you choose for your edge plane normals).

You may also notice that we could have easily solved this problem using the other spanning edge plane shown in Figure 16.59. As you can see, had the edges been tested in a different order, a different spanning edge may have been found first and used to split the polygon into two convex pieces. In this case, the results are different but the overall goal has still been achieved. That is, the two resulting polygons are different from those generated from the edge plane used in Figure 16.58 but the two resulting pieces are still convex. This is not unlike the BSP tree compiler where, regardless of the order in which we choose the split planes, the level will always be broken into convex regions. The shape and size of those regions may be totally different and dependant on the order in which polygons were selected as split planes, but the overall goal of convexity is achieved regardless of the splitter selection order.

Figure 16.59

Of course, it is entirely possible that even after the non-convex polygon has been split by a spanning edge plane, that the two resulting pieces may themselves still not be convex. Therefore, after the non-convex polygon has been split into two child polygons, the same process must be repeated on both of those children to test for convexity. If we find while classifying one of the child polygons against each of its edge planes that a spanning case arises, we know that the child must be split by this plane into two further child polygons. The process repeats itself until we finally test the edges of the child polygons and find no edges which intersect the interior of the polygons.

In Figure 16.60 we see another example of a convex polygon which cannot be resolved into two convex components using a single edge plane split. In the leftmost image we show the original non-convex

polygon and we have highlighted the first edge plane we found that the polygon vertices are spanning. We split the polygon by this edge plane resulting in the two new polygons shown in step 2. The rightmost polygon in step 2 is clearly convex and when we test its edges we find that none of them intersect the interior of the child polygon. However, the leftmost polygon in step 2 does still have edges which intersect the polygon and therefore this child polygon will need to be further subdivided into two new children.

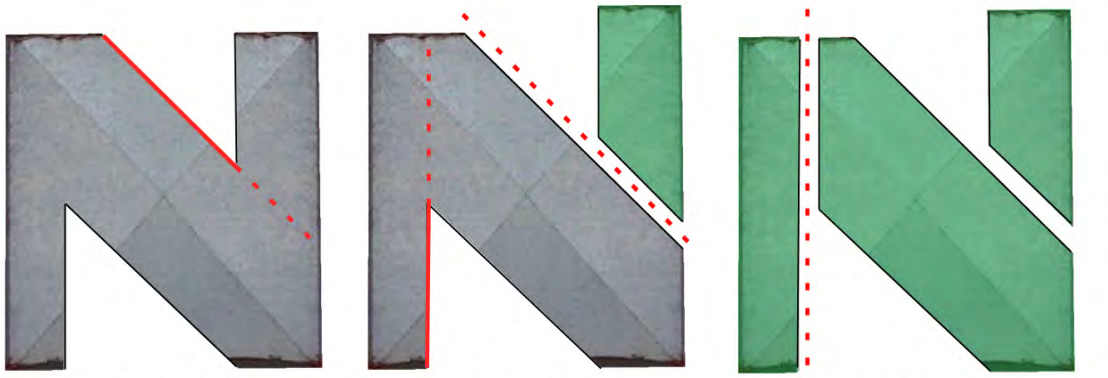


Figure 16.60

In step 2 we highlight the edge in red which we find in the leftmost polygon to be intersecting its interior and once again perform a split. This divides the leftmost child polygon into to new convex polygons shown in step 3. When we test the edges of these children we find that none of their edges intersect the interior space of the polygon to which they belong and thus, we have successfully broken the complex non-convex polygon into three convex ones.

When we examine the images in Figure 16.60 it becomes clear that using the edges of the polygons as the split planes will eventually always carve the polygon up into convex areas. It is a simple process of eliminating all spanning edges until they are removed. A convex object has no spanning edges so when we finally achieve this goal, we have logically eliminated everything from the original polygon that made it non-convex.

What also starts to become clear is that this is essentially how our BSP tree compiler subdivides our level geometry by using the planes of the polygons that comprise the level. For example, if we imagine that the polygon shown in step 1 of Figure 16.60 was actually a top down view through the roof of a building, we can see that the building would be broken up into convex areas using this technique. Our BSP tree compiler does exactly the same thing -- it makes sure that every polygon in the level is used as a split plane and when a plane is found which does intersect the geometry in the level, the geometry is split and assigned to polygon lists that are passed down either side of that node.

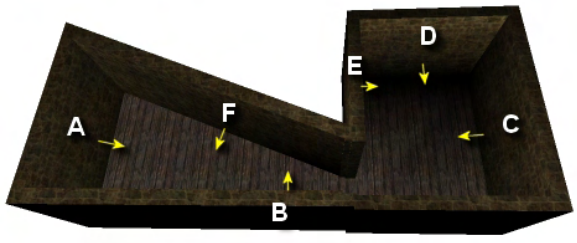


Figure 16.61

In Figure 16.61 we see a top-down view of a piece of 3D geometry. The geometry is a single room, but that room is non-convex. We will now see how the BSP compiler would break this non-convex geometry into two convex leaves. ***Again, each leaf always represents a convex area in the BSP tree.***

> **Note:** In Figure 16.61 and the images that follow the polygons are assumed to all be facing in towards the interior of the room. Back face culling has been disabled to show a better illustration of the geometry being compiled. Furthermore, the polygons in these diagrams have been given some degree of thickness to aid the clarity of the diagram, but as we know, in reality these polygons would be infinitely thin. Finally, although we have shown the room as having a floor polygon, we will ignore this polygon for now to help simplify the number of nodes and operations that would need to be done. The floor is simply there to aid the 3D perspective of the image. We will compile only the walls.

We will now step through the process that would be involved in building a BSP tree using this level. We will use this example to get more comfortable with the idea of passing the polygons down the tree into the leaf nodes during BSP compilation. As this is a fairly simply level to compile we will not accompany each image with a tree diagram. All we are trying to accomplish here is an understanding of why the BSP tree carves up the space of the scene into convex regions. The polygons in these images have been labeled A through F, which indicates the order they are to be selected as splitters during the compile process.

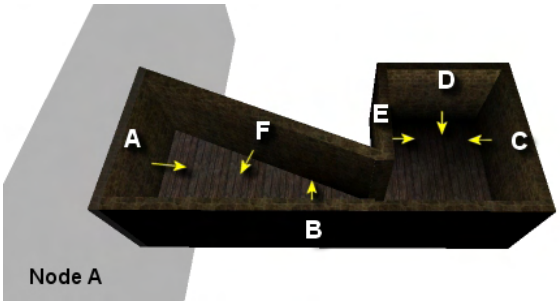


Figure 16.62

In Figure 16.62 polygon A is selected as the root node and marked as having been used as a splitter so that it will not be selected again. All the polygons (A through F) are classified against node A and are found to exist in its front halfspace, so they are compiled into a front list and passed down the front.

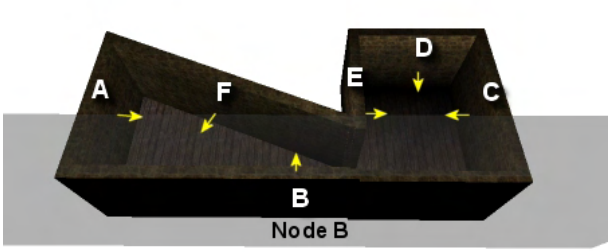


Figure 16.63

Polygon B is selected from the list at A's front child and is marked as having been used as a splitter. All the polygons (A through F) are in node B's front space so are passed down the front.

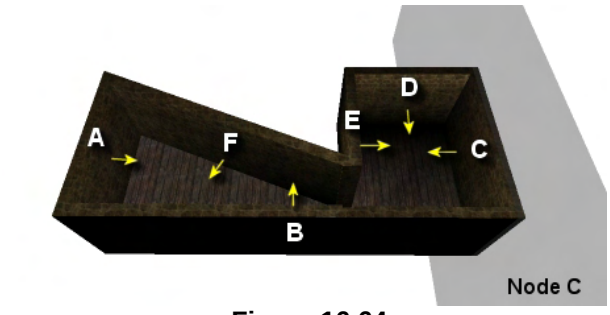


Figure 16.64

Node C is selected next to be the child node of B (Figure 16.64). Polygon C is marked as having been used as a splitter and all the polygons are classified against node C and found to exist in its frontspace. Once again, they are all added to the front list of the current node and sent down the front of node C where a new child node will be created.

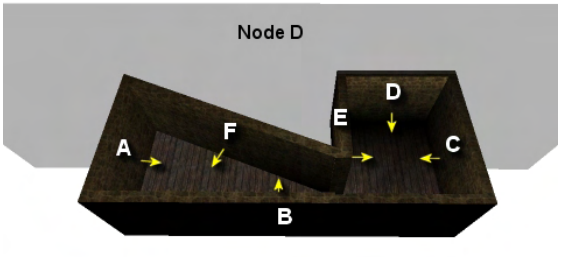


Figure 16.65

In Figure 16.65 we can see that polygon D is selected as a split plane next and the polygon is marked as having been used as a splitter. All the polygons in the list that were passed into node C (polygons A through F) are classified against node D and are once again found all to exist in its frontspace. They are packaged into the front list and passed down the front of node D.

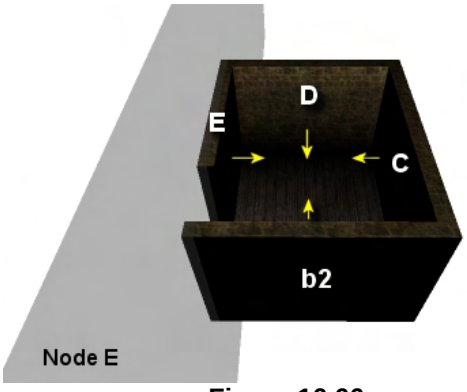


Figure 16.66

Down the front of D the BSP compiler chooses polygon E as the next split plane. This is the interesting bit. Plane E is the plane that intersects the interior of the geometry and therefore the geometry in the list will need to be split. When the polygons are classified against node plane E, polygons E, D and C are found to exist in its frontspace and are added to the front list, while polygons A and F exist in its backspace and are therefore added to the back list. Polygon B however is spanning the plane and is split into fragments b1 and b2 with b2 being the fragment located in the front halfspace of node B. b1 is added to the back list and b2 is added to the front list. What is vitally important to know is that when polygon B, which has already been used as a split plane, is split, we carry over the status of its 'BeenUsedAsSplitter' Boolean into the child splits b1 and b2. This will stop b1 and b2 from being used as split planes later since their parent polygon has already been used. Otherwise we would have redundant split planes which would create more nodes in the tree.

The front list at node E will contain polygons E, D, C, and b2 and these will be passed down the front of node E as shown in Figure 16.66. When we get down the front of node E however, we find that we have no more polygons that have not yet been used as splitters and therefore, our job is done. At this point we must have a group of polygons whose planes represent a convex volume and therefore we have determined that there is a leaf in front of node E. Furthermore, by clipping and passing the polygons down the tree during the build process, we have also collected the polygons for this leaf at the same time. As you can see in Figure 16.66, the first leaf we create is comprised of four polygons which all face into the center of the leaf. If we query the tree for a position and find that the query position is located in this leaf, we now it is located in the section of the original room shown in Figure 16.66.

Note: A leaf is created whenever we have a list of polygons passed into a node have all been used as splitters. Unlike the quad-tree, oct-tree and kD-tree which can have many stop codes which determine when a leaf is created, with the BSP tree we keep creating child nodes until we find that a node is passed a list of polygons which have all be used to create split planes. No other stop code is necessary. As soon as no split candidates remain, a leaf is created and the planes of the list polygons passed into that node are guaranteed to form a convex region that bounds the leaf.

With the leaf in front of node E created we must now process node E's back list which contained polygons A, b1, and F. When we step down the back tree of node E we find that only one polygon remains in the list which has not yet been used as a split plane. This is polygon F which is used to create the back child node of E. Polygons A, b1 and F are classified against node F and are all found to exist in the front space. However, as polygons A, b1 and F have all been used as splitters we do not create a new front child for node F but instead attach a leaf structure to its front containing these three polygons. We now have our second convex leaf. Our level as now been divided into two convex leaves as shown below in Figure 16.68.

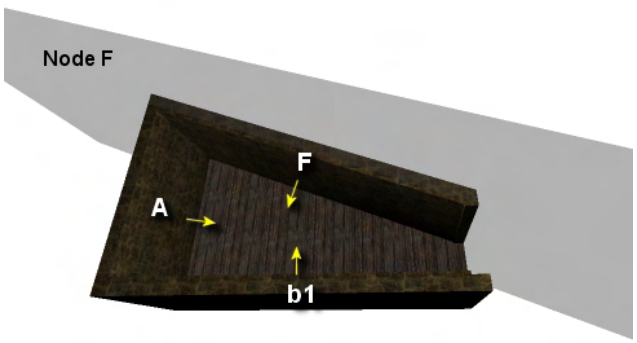


Figure 16.67

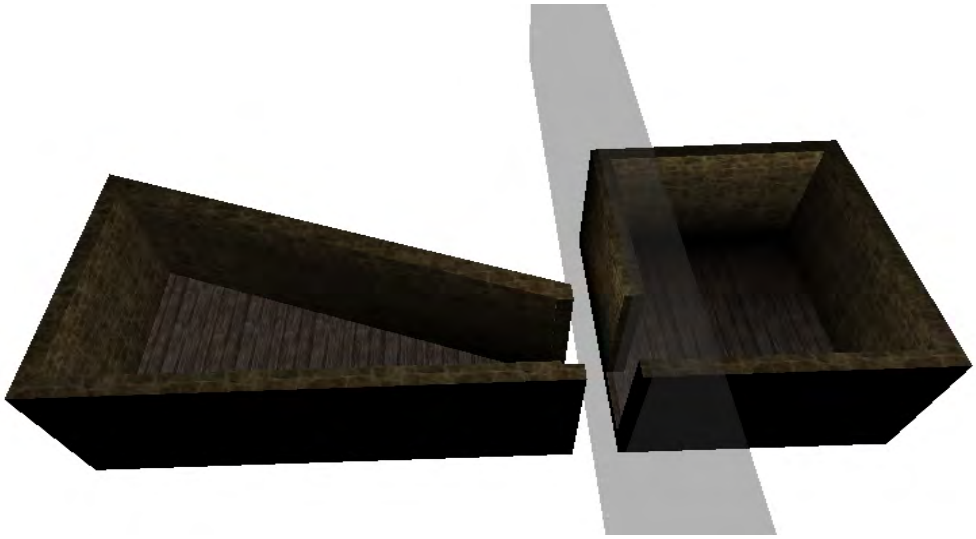


Figure 16.68

We have now seen exactly why the BSP tree compilation technique breaks the world into a series of convex areas and we have also seen how much it is like the polygon examples we examined earlier. In this case, it was node E's plane that was found to be the spanning plane and responsible for dividing the initial polygon list into two child lists in front and behind the split plane. We have also seen how a leaf is created whenever we reach a point in the recursive process where every polygon in the list passed into a node has already been used as a splitter, making it impossible for another node plane to be selected. It is also worthy of note that whenever no polygon data gets passed down one side of a node, an empty leaf is created there. Figure 16.69 shows the final resulting planes of the BSP tree that was just compiled and we can see that it has carved the world up into seven convex leaves. Only two of the leaves contain polygon data and those are the only two leaves that have all their bounding plane normals facing into the interior of the leaf.

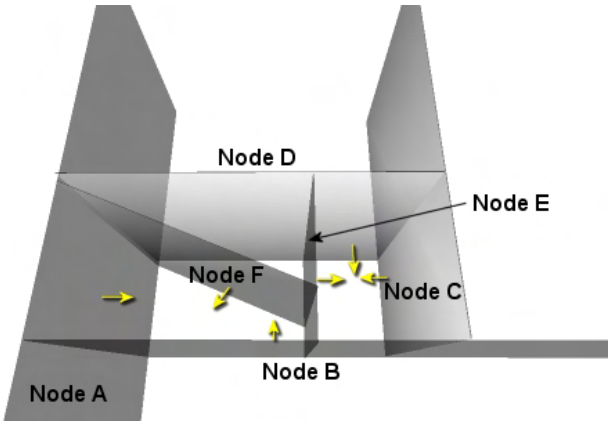


Figure 16.69

16.3.3 Solid / Empty BSP Trees

The title of this section might lead you to believe that we are going to discuss a different type of BSP tree that can encode solid and empty information. However, the current tree we have will already do that if we send it geometric data in an ordered and sensible format. Providing that we send the compiler geometry that obeys certain conditions (i.e., *legal geometry*), a level will be compiled such that polygon data will only ever be stored in front leaves. Back leaves will always be empty of polygon data and therefore, represent areas of solid space within the game level.

When we refer to a BSP tree as a 'solid' BSP tree, it simply means that because of the way the input data has been constructed, we can easily deduce which areas of the scene are empty of obstructions and which areas of the scene are assumed to be solid space (e.g., the space in the middle of a brick wall). Solid space is space that the player cannot see through and is a region where the player should never be allowed to move. We will see shortly that assuming we send the geometry to the BSP compiler in the correct format, every leaf attached to the back of a node will represent a solid area (i.e., a leaf of solid space). Every leaf attached to the front of a node will represent an empty space leaf. That is, a leaf in which the area of space is assumed to be empty and that the player can both see and navigate through. The polygons of the level will always be assigned to empty leaves (front leaves), which may sound strange at first as we normally consider the polygons to be the solid objects that we cannot navigate through. This is still the case of course in theory. However, as discussed earlier, the polygons will always lay on the boundary planes of a leaf and therefore, it is the empty space between those polygons in which the player is allowed to navigate. Furthermore, because of the way we handle the on-plane case during BSP tree construction, polygons will always end up in leaves which their normals face into. This means, when the player is located in empty space, the polygons assigned to that leaf will be facing into the leaf, and therefore, facing into the same halfspace as the player.

Although this might all sound a little abstract (until we look at some examples of geometry that will compile a solid BSP tree), assuming that this is correct, why is it important that we have access to this solid/empty space information? The reasons are actually numerous and quite important.

First, if we can guarantee that every back leaf represents solid space, then performing line of sight tests becomes trivial. If we wish to determine whether point A can see point B, then we would create a ray from point A to B and send it through the tree. At each node the ray would be sent down the front or back of the tree depending on the ray/plane classification result at that node. If the ray is found to span the plane then the intersection point C with the plane is calculated and the ray is split into two by creating two child rays A->C and C->B. These two ray fragments are passed down their respective sides of the node. This process continues until the ray fragments pop out in the leaf nodes. We performed an identical process to this in lesson 14 when we discussed passing a ray through the kD-tree (CollectLeavesRay). However, for a line of sight test, if at any point we find that a fragment of the ray has been passed into a back leaf (i.e., a region of solid space), we know that no line of sight can possibly exist between these two points. That is because part of the ray passes through an area of space that is solid, such as a section of wall or floor for example. When this is the case we can immediately return false for the line of sight test. Performing line of sight tests in this way is much faster than testing each polygon for intersection with the ray. Using the BSP tree, we have no point in polygon tests to perform at all.

Another important reason for needing solid and empty space determination will become clear when we calculate a potential visibility set in the following and final lesson. A potential visibility set is a block of data that describes to us which leaves are visible from any other leaf within the tree. If we have a potential visibility set at our disposal, we can simply traverse the tree each frame to find the leaf in which the camera is currently located and fetch the visibility set for that leaf. This will instantly tell us which leaves are visible from the current leaf we are in and we can render each of these leaves. The potential visibility set will not calculate the visibility information based on leaf inclusion within the camera frustum, as with our previous rendering systems. Instead, it will be calculated at development time and will take occlusion into account. This means that if a leaf represents a small room with a small open doorway, the only polygons visible to that leaf will be the polygons forming the walls, floor, and ceiling of the room itself and the small number of polygons that can be seen through the open doorway from within that room. All other leaves that lay behind the walls will not be in the leaf’s visibility set and thus we typically render only a small number of polygons and reduce overdraw significantly. Essentially, instead of rendering everything that is inside the frustum, we render only what is actually visible from that location. Polygons that are located well within the frustum but are occluded by nearer geometry will not be in the visibility set for that leaf and will not have to be processed in any way. The potential visibility set for each leaf takes a very long time to compile (sometimes many hours) so it is done as a development time process. The information is then saved out to disk and loaded into the game at runtime and used. In the following lesson we will write a PVS (Potential Visibility Set) calculator and one of the chief pre-requisites for being able to calculate the PVS for a game level is that the level be compiled into a solid BSP tree. Take a look at Figure 16.70 to see why this is the case.

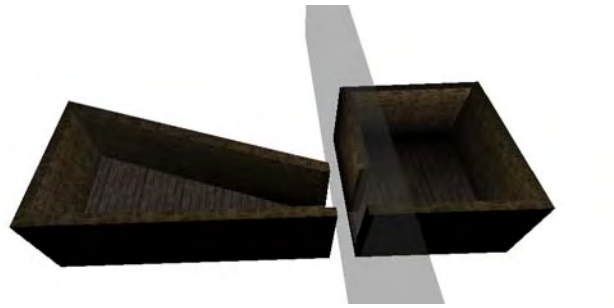


Figure 16.70

Here we see the two leaves that the non-convex room was broken into in our previous example. In this image we have deliberately separated the leaves so that we can see the gaps in the polygon data were the two leaves connect. We can think of this gap as the doorway between the two leaves. The plane that created the split into two leaves is also shown. You can imagine how in a really complex level thousands of leaves will be created which all have similar doorways that lead into one another. In order to calculate the visibility for a leaf, we will need to know the size of each of these doorways so that we can build view volumes out of them which will describe everything that a leaf can see through its doorways (that lead into other leaves). Do not worry about how this is done as this will be the entire focus of the following lesson. The problem we have is that we need to know the size of these doorways which is information we currently do not have. We know the planes on which these doorways will lie because they are the node planes that split the geometry (the gray plane shown in Figure 16.70) but we do not know the size of the doorway. We will need to build temporary polygons (called portals) that will fit these doorways exactly. Once we have calculated all the portal polygons for a given leaf we will have created polygons that fit these doorways. We will then be able to construct view volumes from these portals which will describe what can be seen from the leaf, through those portals, and into other leaves. This will tell us which other leaves are visible from the current leaf having its visibility set calculated. If we know which areas of the world are solid space (such as located in a wall for example) and which areas are empty, we can build an initially huge polygon on that split plane and send it through the BSP tree. Any fragment of the polygon that ends up in a solid leaf (a back leaf) can be deleted as it is obviously situated inside a solid object or is outside the exterior walls of the level. At the end of the process we will end up with a polygon fragment that ends up in empty space and this is the portal(s) to the leaves that were subdivided by that split plane.

The calculation of a PVS will be discussed in the following lesson but what has become very apparent is the need of the PVS calculator to be able to determine during the portal creation phase whether or not a portal fragment has ended up in solid or empty space. Thus, we need to make sure we provide the PVS calculator with a BSP tree which has been compiled with geometry such that every back leaf can be considered solid space and every front leaf can be considered empty space. This is all about the geometry we send into the tree and not about changing any of the BSP compiler code in any way, as we will soon examine.

Another reason we will need to be able to compile a solid tree will be when performing CSG operations which will be discussed in the final section of this lesson. CSG operations allow us to carve one object from another or union two objects together into a single mesh. We will see later that the union operation is especially important to us as it allows us to fuse all the meshes comprising the level into a single static mesh which can be compiled into a solid BSP tree. The union operation will remove hidden surfaces, which are essentially polygons from one object embedded (or partially embedded) inside the interior space of another object. Geometry that has such arrangements of polygons is considered illegal geometry as it breaks the rules laid down by the solid BSP tree. The tree compiled from such data will contain invalid solid/empty space information thus causing PVS calculation and any future CSG operations on that geometry to fail.

Note: Such data is referred to as 'illegal geometry' from the BSP tree's perspective and will result in an invalid BSP tree. We will discuss illegal geometry in some detail later on in this lesson and will discuss ways to correct such problems in the geometry. CSG operations allow us to remove these hidden surfaces that would cause the BSP tree created to be invalid and is another process that relies on solid/empty information.

So we have seen that the need to be able to build a BSP tree which contains solid/empty information is important especially to future processes that we will undertake. We have also learned that a solid BSP tree is fundamentally no different from a normal BSP tree with respect to the code -- it all comes down to the geometry that we send our compiler. If we send legal geometry to our BSP compiler, a level will be created which will always have empty back leaves and polygons stored in the front leaves. Those back leaves will always represent solid space and the front leaves will always represent empty space.

This may all seem incredibly hard to imagine at the moment. Indeed it seems odd that by just supplying geometry constructed in a certain way we will get this solid/empty information for free. However, it is absolutely true and is the exact reason that artists that are developing scenes for a game engine that uses BSP/PVS technology will use world editors such as WorldCraft™ or GILES™. These editors allow the artist to construct worlds from a series of simple solid primitives using CSG operations to carve and union these simple shapes into more complex objects. The editors force (for the most part) the artist to build scenes that are considered legal geometry just by the very way that they are constructed. This will all make more sense when we cover exactly what solid/empty space is and what is considered legal/illegal geometry by the solid leaf BSP compiler.

In our node tree discussion we used an example level that was extremely unrealistic and made reference to the fact that because a polygon lay on of the node planes, it may actually lie on the boundaries of a front and back leaf. Figure 16.71 highlights such an example from this level. We can see for example that polygon B has been assigned to the leaf in its node plane's frontspace and therefore this polygon would be assigned to leaf 1. However, notice that this polygon (and its plane), also form a boundary for the leaf that lies behind the plane. If our camera was located in the leaf bounded by planes A, B and c2 for example, we would be looking at the back of polygon B, which would be back face culled. This would allow us to see right through this polygon into leaf 1 which would completely destroy the visual integrity of the level.

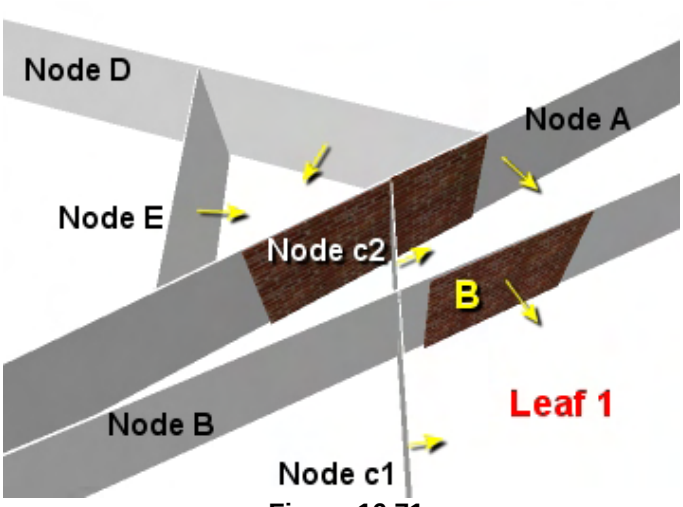


Figure 16.71

Of course, this never happens in a real game since our artists would not include a wall that can be viewed from all sides as a single polygon. As Figure 16.72 shows, if we were to represent a section of wall as a single polygon it would only be visible to the camera when viewed from its front side. If the player was allowed to navigate the camera such that it could view the polygon from behind, it would be back face culled and would therefore be invisible.

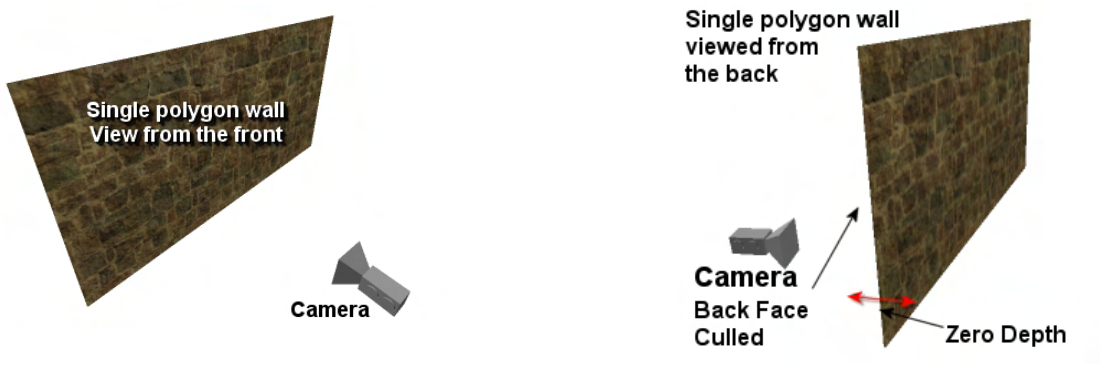


Figure 16.72

Representing a wall in such a way would only be acceptable if that wall formed one of the exterior walls of your interior scene and as such, the camera could never navigate around the back of it. In fact, even when viewed from the side or top we compromise our belief that this is supposed to be a solid wall as it would become clear that this polygon is nothing but a thin sliver of texture.

World editors such as GILES™ aid in helping the artist avoid such mistakes by forcing the placement of solid objects. For such a wall section, a cube would be placed and scaled to fit the desired dimensions. The cube has 6 faces instead of 1 and can be viewed from all sides. Regardless of the side you are viewing the wall from, there will always be faces with normals oriented towards the view direction. This obviously means that 6 faces get added to the scene instead of 1, but it also means that we now have a wall that is not only viewable from all directions, but also has a thickness when viewed from the sides, above or beneath, as shown in Figure 16.73.

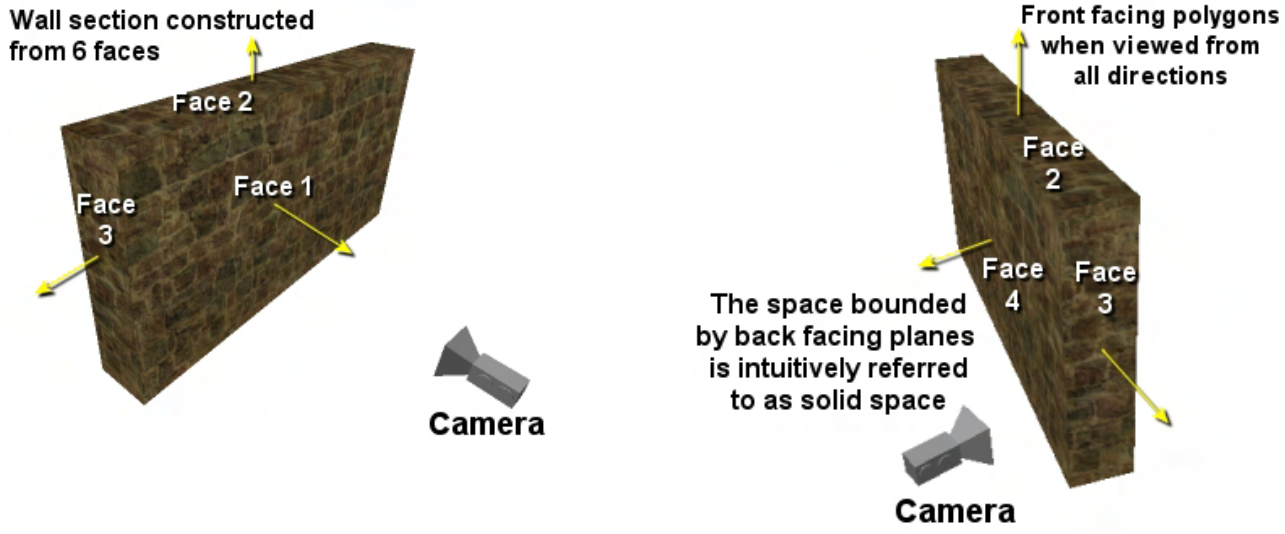


Figure 16.73

Assuming that this wall was the only geometry in our scene, let us see what happens when we compile it into a BSP tree. For the rest of this example we will simplify to a 2D top down perspective and will ignore the top and bottom faces. That is, we will assume that we are compiling a 4 faceted cube instead of a 6 and are viewing that cube from above.

Figure 16.74 shows the compilation of this 2D wall section (the cube). Polygons pA, pB, pC and pD are the outward orientated faces of the wall and the gray arrows show their face normals. The polygons are assumed to be selected as splitters in alphabetical order during the build process, creating node planes A, B, C and D. To the right we see the BSP tree that is generated from such data.

As you can see, node A is selected first and all of the polygons are classified against it. pA is co-planar and same facing which means it is sent down the front of node A. As the only polygon in the front list is polygon pA, which has already been used as a splitter, a leaf is created and polygon pA is assigned to it. The rest of the polygons are added to the back list and sent down the back of node A.

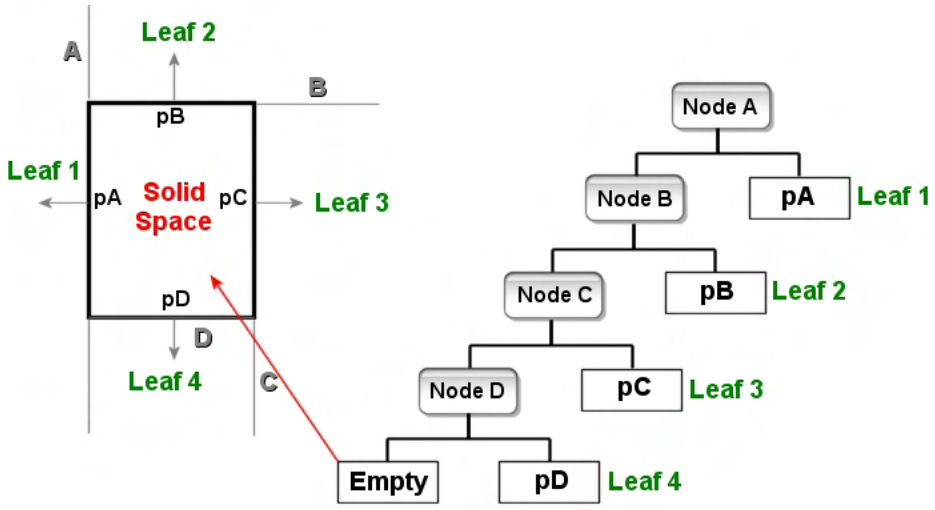


Figure 16.74

Down the back of node A polygon pB is selected as the next splitter which creates node B. Polygons pC and pD are added to the back list and polygon pB is co-planar and same facing which means it gets added to the front list. As there is only one polygon in the front list for node B (polygon pB) which has already been used as a splitter, a new leaf (leaf 2) is created down the front side of node B and polygon pB assigned to it. As we continue down the tree we see that polygons pC and pD end up getting assigned to their own leaves down the front of nodes C and D respectively. At node D however, there are no polygons in the back list so an empty leaf is created. As Figure 16.74 shows however, this empty leaf represents the area of solid space behind each of the polygons.

> **Note:** Whenever we compile a convex object with outward facing normals into a BSP tree we get a one sided tree as shown in Figure 16.74 where the nodes are all back children. If we were to reverse the plane normals so that all the polygons faced inward, empty space would be in the middle of the cube, solid space would be all around the outside of the cube, and the tree would be a one way tree down the front side instead of the back.

When we examine Figure 16.74 it starts to become clear why we can rely on a back leaf always being empty and representing solid space. When a polygon is co-planar and same facing, it is passed down the front of the tree, which means it will always end up in a leaf down the side of the tree which its normal is facing into. If the polygon is co-planar but not same facing, it is sent down the back list where it will later be used to create its own node. The polygon is obviously going to be found to be co-planar and same facing with this new node so is passed down its front. Thus, every polygon will eventually get added to a front leaf. Conversely, because this is the case, no polygons will ever be added to back leaves. Since all polygons always end up facing into the empty space of the leaf to which they are

assigned, the solid leaves represent the space behind those polygons and thus will never contain polygons of their own. Examining Figure 16.74 again, we can see that if we pass a query position through the tree that ends up in this empty back leaf, it will be located behind polygons pA, pB, pC and pD and therefore be located in center of the wall (i.e., in solid space).

Although this is probably about as simple a level as you could possibly compile, we will examine more complex levels later and show that the same holds true for our entire scene as long as we obey certain level creation rules. That is, provided we supply the BSP compiler with legal geometry, every solid region in the scene will be represented as a back leaf and empty space will always be represented as a front leaf. We will examine exactly what constitutes illegal geometry a bit later and see why it causes the solid/empty space relationship to break down.

The addition of solid and empty information to our BSP leaf tree is the primary reason behind the importance of ensuring that the on-plane case is handled correctly during construction. Recall that if a polygon is co-planar with the current node, then we add that polygon to the front list and ensure that it is not used as a splitter again. That is, we flag this polygon as used *only* if it points in the same direction. If a polygon's vertices are co-planar but its normal points in the opposite direction, then we add it to the back list, but we **do not** remove it from consideration as a future node plane candidate. This will ensure that the polygon will be selected as a split plane later, which is vitally important to the solid/empty integrity of our level

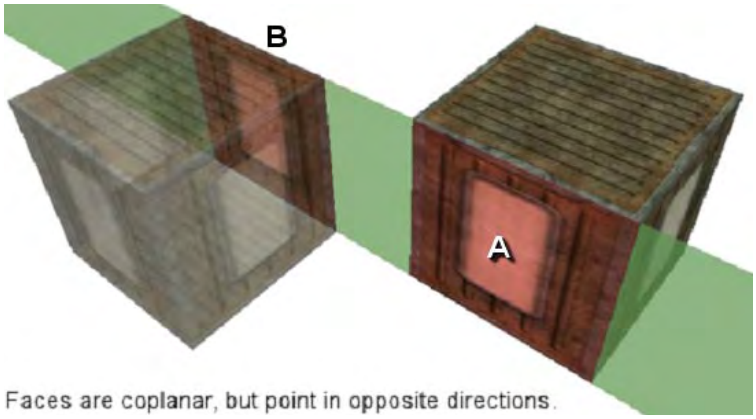


Figure 16.75

Consider the situation outlined in Figure 16.75. The example shown produces an arrangement unlike that of any of our previous test geometry. Here we have a scenario in which we find two polygons that lie on the same plane as one another but point in opposite directions. Imagine that we were to select the closest polygon as the candidate for the separating node plane depicted here. When we compare the polygon list against this plane, this candidate polygon is added to the front list as expected. Due to the fact that the co-planar polygon that is furthest away from the camera faces into the back halfspace of the node plane, it is added to the back list. As discussed in the previous section, we would not usually mark the co-planar polygon that was added to the back list as used. Yet, it may not be clear at this point exactly why this particular step is so important.

Contrary to our current building concept, if we were to remove **both** of these polygons from consideration by flagging them as used at this node, we would end up with a tree structure similar to that shown in Figure 16.77. Even though we are focusing on one particular node here and showing only the subsection of the tree that would be generated by the two cubes in the above example, we can clearly see that if *both* polygons 'A' and 'B' were flagged as used when node A was created, we would end up with a situation in which polygon 'B' has been added to a leaf that falls behind the node.

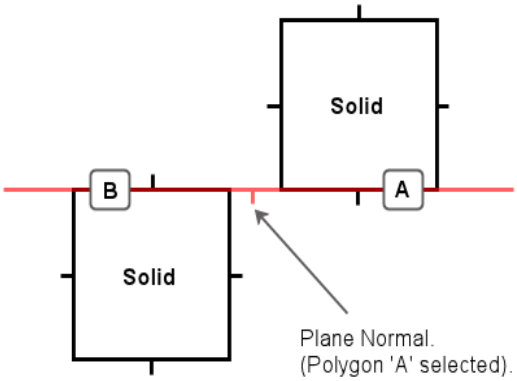


Figure 16.76

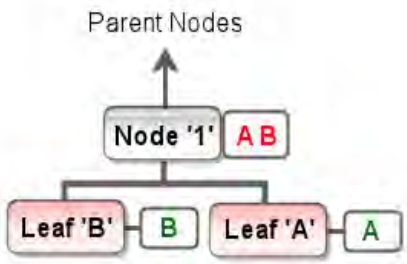


Figure 16.77

Note: Figures 16.76 and 16.77 do not represent a full and valid BSP leaf tree. These diagrams are merely intended to draw your attention to one particular portion of a larger tree.

In this particular example we can observe that the front side of polygon ‘B’ faces into a leaf contained in the back halfspace of the node. This conflicts with the key property that allows us to identify solid and empty areas in our tree because we now find that a leaf which should clearly describe empty space falls *behind* a node plane.

This is the reason why it is vital that we implement the on-plane case in the manner discussed in previous sections. By **not** marking polygon ‘B’ as ‘used’ during the classification step for the first node, we allow our compiler to select that polygon for node creation at a point further down the tree structure. As we can see in Figures 16.78 and 16.79, in this case we have allowed the compiler to generate a new node for each polygon which points in opposing directions.

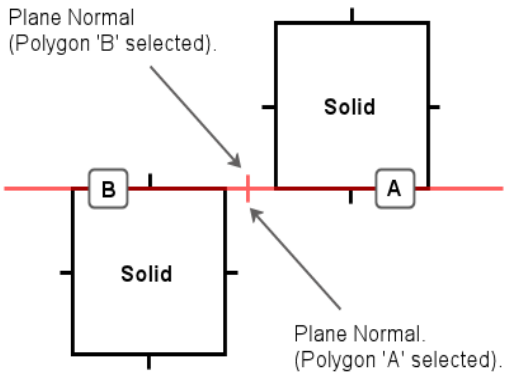


Figure 16.78

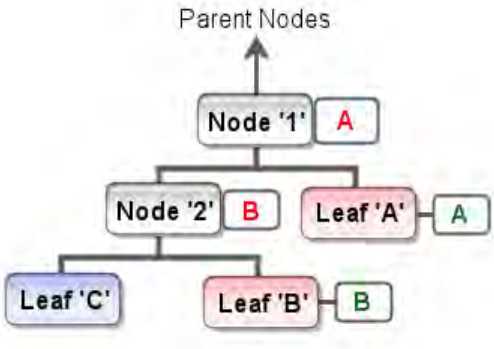


Figure 16.79

Once we allow polygon ‘B’ to create its own node, we can see that we have restored the ability to classify a leaf as solid if it is contained within the back halfspace of its parent node, or empty if it exists in front. Notice that polygons A and B both eventually end up getting assigned to front leaves.

Referring back to our single wall section example again and assuming that the wall section we have just compiled into a BSP tree is currently the only geometry in the scene, you will see that the BSP tree automatically provides us with the ability to perform line of sight tests in an extremely efficient manner.

In Figure 16.75 we show two positions that exist in empty space labeled J and K. Position J is located in leaf 1, where the wall polygon pA could be viewed. Position K is also in empty space (leaf 3) and from this position the wall polygon pC can be viewed. However, position J cannot see position K because the wall polygons are in the way.

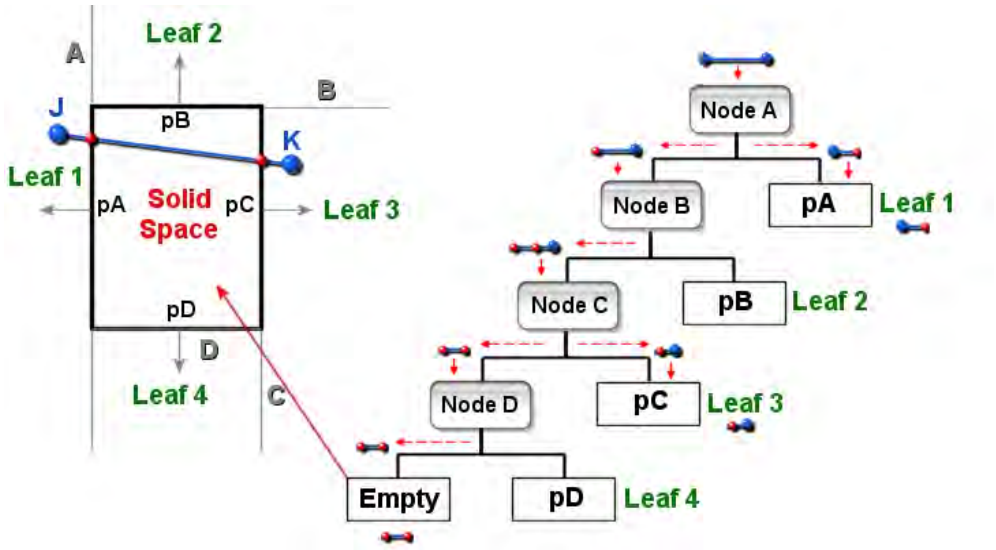


Figure 16.80

Usually, to determine if line of sight exists between points J and K we would have to perform expensive ray/polygon tests between the ray and each polygon in the vicinity of the ray. This could be a very large number of expensive tests in a complex level. However, we can see in this simple example that we can test very efficiently if line of sight exists between two points in a solid BSP tree by simply sending the ray down the tree and returning false as soon as a fragment of the ray ends up in solid space. If all fragments of the ray end up in empty space then a line of sight does exist. Note that this is determined without testing the actual polygon data.

As shown in Figure 16.80, the ray is sent in at the root node. A ray/plane test determined that the ray spans the node, so the intersection point with the plane is calculated. That ray is then split such that this intersection point becomes the end point for the child in one halfspace and the start point of the child in the other. In this example, the front split of the ray is sent down the front of node A where it arrives in Leaf 1, which is empty space. This is the section of the ray shown in Figure 16.80 between point J and the red dot on the plane of node A. The back split of the ray is passed down the back of node A where it is classified against node B. The ray fragment is completely behind node B so is passed down the back into node C. At node C the ray is found to span the plane and is split, creating two child rays. The ray fragment in the front halfspace of node C is shown in the diagram as the segment starting at the red dot on node C and ending at point K. This is sent down the front of node C where it lands in leaf 3 which is empty space. The ray fragment that was found to lie in the back space of node C can be seen in the diagram as the line that joins the two red dots on nodes A and C. This is passed down the back of node C where it arrives at node D. It is found to lay in node D's back halfspace and is passed down the back of node D into a back leaf. Since the ray has entered a back leaf this must mean that this fragment is in solid space (as we can clearly see in the top down view of the cube). Therefore, we return false from this node, and eventually the function, indicating that no line of sight exists between these two points.

It is important to understand that just because we want to have a solid BSP tree does not mean that every wall, floor or ceiling in the level has to be constructed using six sided cubes. It simply means that the geometry must be constructed such that empty space is always bounded by front facing polygons and

solid space is always bounded by back facing polygons. For example, if we were to invert the normals of the polygons in the previous example and compile a BSP tree, we would get a different, but still perfectly valid, tree.

Figure 16.81 shows the result of compiling the same polygons but with negated normals. Now the polygon normals all face into the center of the cube which creates only one empty leaf. Now, there are four back leaves behind nodes A, B, C and D which combined represent the solid space around the outside of the cube. This is still a perfectly valid solid tree. In fact, we might imagine that the inward facing polygons represent the walls of a room and leaf 1 is the empty space inside that room in which the player is allowed to walk. This is still perfectly valid because empty space is still bounded by inward facing polygons and solid space is still separated from empty space by the backs of polygons.

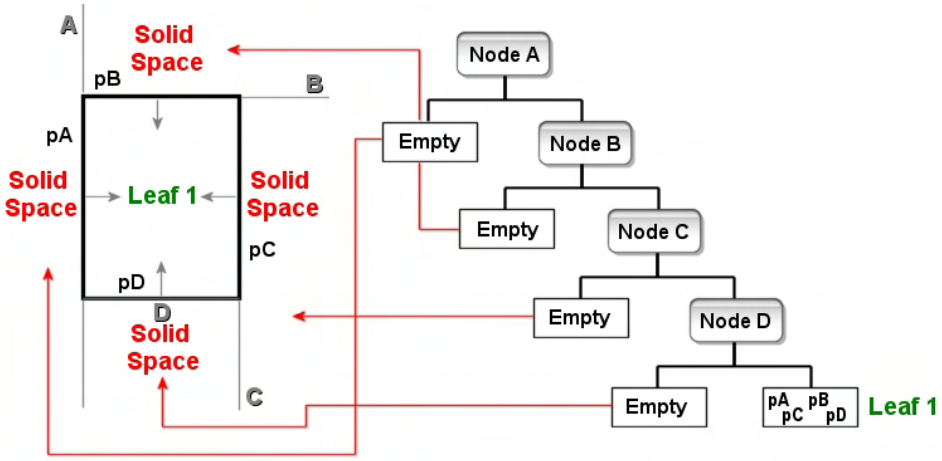


Figure 16.81

To illustrate the fact that our BSP tree code has not changed in any way, let us compile the non-convex room we looked at earlier. Remember when looking at Figure 16.82 that the walls of the room are supposed to be single inward facing polygons. In this diagram the walls have been given artificial thickness to better aid the visual demonstration. Back face culling has also been disabled so that polygon B is still rendered even though we are technically viewing it from its back side.

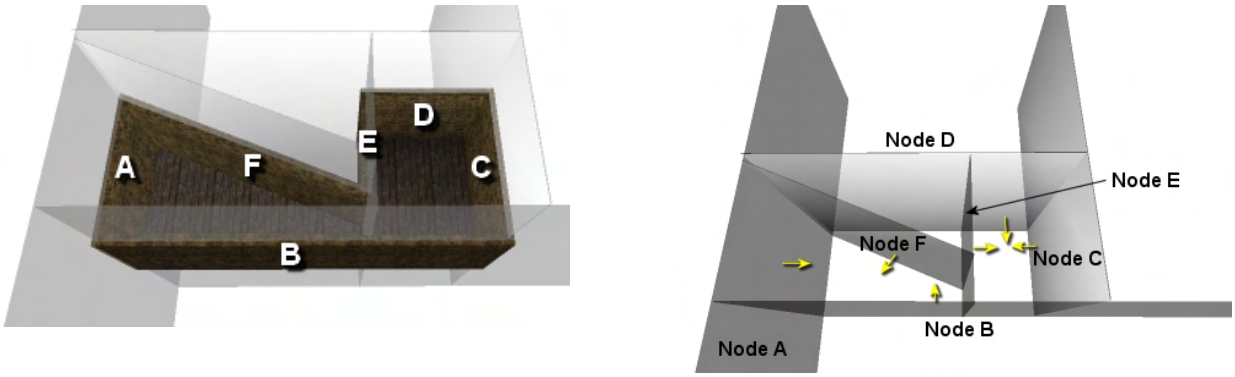


Figure 16.82

The leftmost image shows the original polygons that were used to compile the BSP tree. The letters assigned to them describes the order in which they were used to create node planes. Since we are now very familiar with the BSP tree building process, we will not show the construction process of the tree. The rightmost image shows the node planes that were created during compilation. The plane normal directions are depicted by the yellow arrows. Again, to keep the tree simple, we will pretend the floor polygon does not exist and has been rendered here only to aid with the visual component of the diagram.

Notice in the rightmost image that we have also created no node for this floor polygon (this is just a choice to keep the tree as small as possible during the examples).

Although you did not necessarily know it when we first examined this piece of geometry, this is perfectly legal geometry that will compile into a solid BSP tree. In this example, we will once again demonstrate our method using a two step approach. We will assume that the nodes of the tree were generated in an initial pass and that the polygons were then passed down the tree and collected in the leaf nodes in a second pass. The geometry depicted here obeys all the rules for keeping solid and empty space separate. The empty space is the space within the room itself and solid space is assumed to be all around the outside. Assuming that the BSP tree has already been compiled, let us now send the polygons (used to create the nodes in the first pass) into the tree and see where they end up. If this geometry is legal, we should find that all the polygons get assigned to leaves down the front of nodes and all leaves created behind nodes should represent solid space. This will also be our final example of populating the BSP tree with polygon data in a second pass. It will allow us to solidify exactly what happens to each polygon as it is passed down the tree before we merge this process into the node construction process and write our final solid BSP leaf tree compiler.

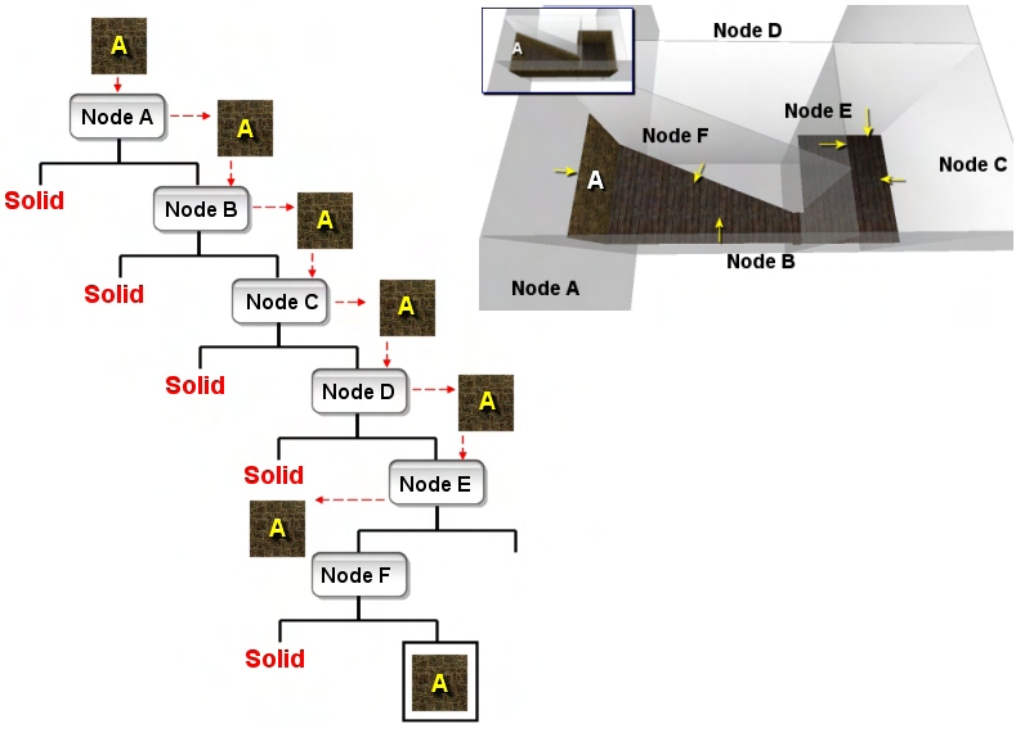


Figure 16.83

As Figure 16.83 shows, polygon A is sent into the tree first. At this point, you should be able to understand why the nodes of the tree are arranged in the order in which they are depicted. Polygon A is classified against node A where it is found to be co-planar and same facing and is therefore passed down the front into node B. We find that the polygon is also contained in the front halfspace of nodes B, C and D so it gets passed down the front at each step until it gets to node E. When classified against node E it is found to exist in its backspace and is passed down the back where it enters node F. As we can see in the diagram, polygon A is located entirely in node F's frontspace so is passed down the front of F. As no

more nodes exist down the front of node F it must mean that a leaf exists here to which polygon A is assigned.

The next polygon we pass down the tree is polygon B. At node A it is found to exist in the frontspace so is passed down the front into the node B. At node B it is obviously found to be co-planar and same facing (as this is the polygon that created this node) so is passed down the front into node C.

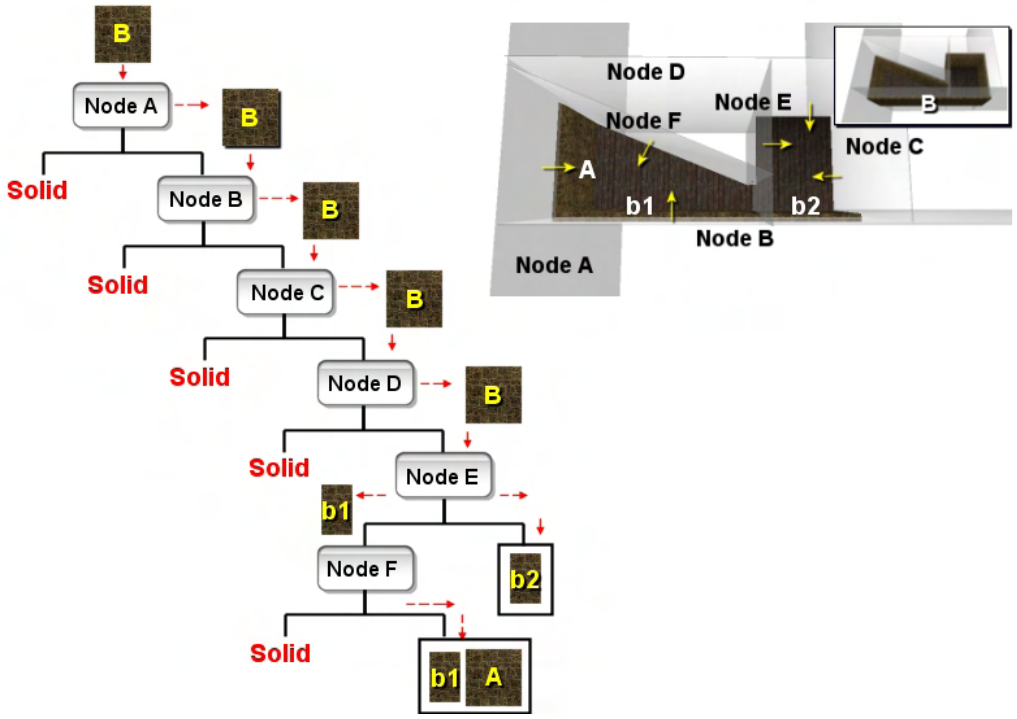


Figure 16.84

Looking at the diagram we can see that polygon B is also contained in node C's frontspace so is passed down the front into node D. Once again, node D has polygon B contained in its frontspace, so polygon B is passed into the front child of node D, which is node E.

At node E polygon B is found to be spanning the plane and is therefore split into polygon fragments b1 and b2. The original polygon B is deleted. Polygon b2 exists in the frontspace of node E so is passed into its front child. As no front child exists however this must mean that polygon b2 has entered an empty leaf attached to the front of node E. Polygon b1 is passed down the back of node E where it arrives at node F. Polygon b2, when classified against node F is found to be contains entirely in its frontspace and is therefore passed down the front of F where it gets added to the leaf that exists there. Polygons A and b1 now both exist in the empty space leaf attached to the front of node F.

Next we pass polygon C into the tree.

As Figure 16.85 clearly shows, as polygon C is passed down the tree it is found to exist in the front halfspaces of nodes A, B, D and E. This means it eventually gets added to the leaf attached to the front of node E.

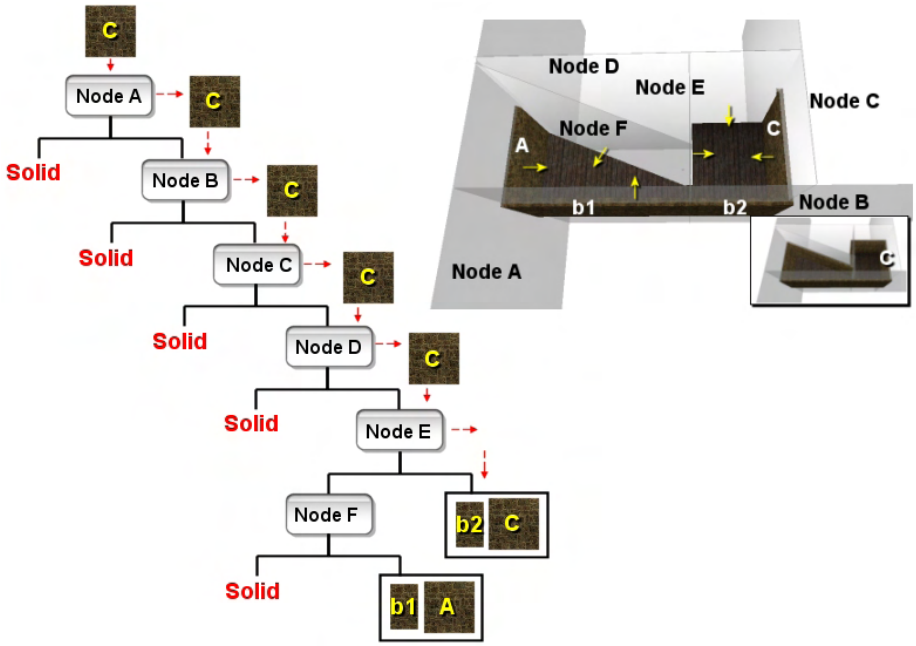


Figure 16.85

Things are looking very good at this point. No polygons have been added to back leaves and so far every polygon we have added has been added to one of two leaves. This is in keeping with what we discovered about this level when we discussed the convexity properties of the BSP tree. You will recall we used this example geometry and showed how the BSP tree would carve the geometry up into two leaves. This certainly seems to be the case. This does not mean that the tree only has two leaves; it means that it will only have two leaves that contain geometry (i.e., two leaves that are situated in front of a node).

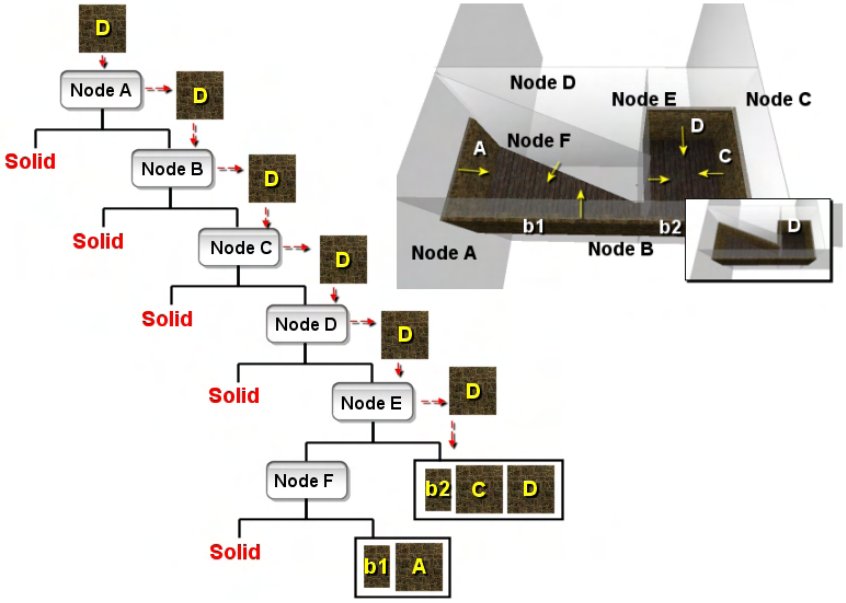


Figure 16.86

Passing polygon D down the tree we see an identical route being taken. By looking at the geometry in Figure 16.86 we can clearly see that polygon D is in the frontspace of planes A, B, C, D and E which results in the polygon being added to the leaf in the frontspace of node E. In the tree diagram we can see that this is the exact classification that happens when polygon D is passed down the tree. Polygon D is added to a front leaf which already contains polygons b2 and C.

Figure 16.87 shows the route that the penultimate polygon (E) takes though the tree. It is classified in the frontspace of nodes A, B, C, D, and E. Of course, node E is the node that was created from this polygon during the building phase and as such, we know it will get passed down the front of such a node. This also clearly demonstrates how this forces polygon E to

eventually be assigned to an empty space leaf that the node plane normal (and the polygon normal) is facing into.

We can see at this stage that the front leaf of E is complete and contains all the polygons that bound that leaf and face into it. All the polygons for this leaf are shown in the square inset in Figure 16.87.

Finally we have polygon F to send through the tree whose route and eventual leaf placement is depicted in Figure 16.88.

Polygon F is found to be in the frontspace of nodes A, B, C and D and is therefore passed down the tree into node E. At node E the polygon is found to be in the backspace of the node, so is passed down the back of E into node F.

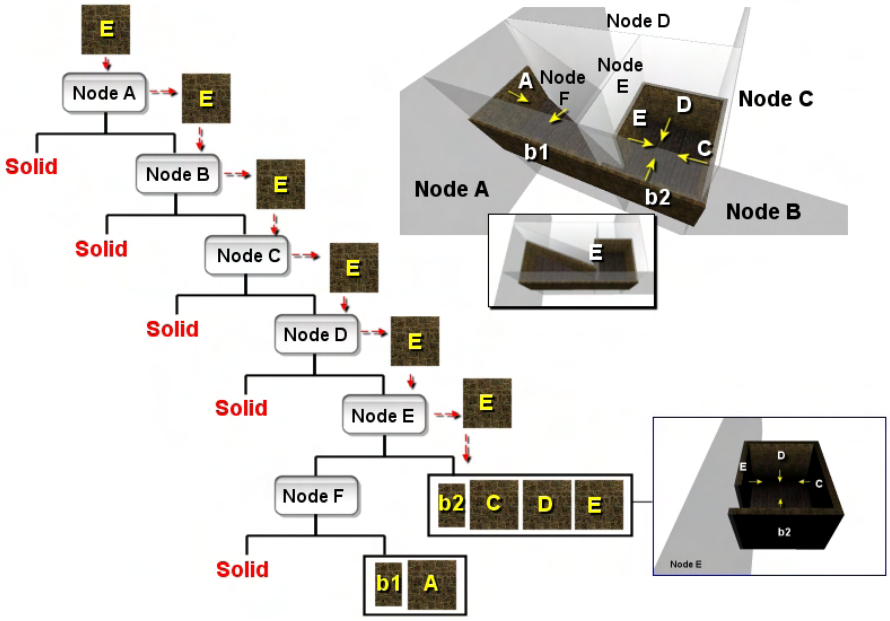


Figure 16.87

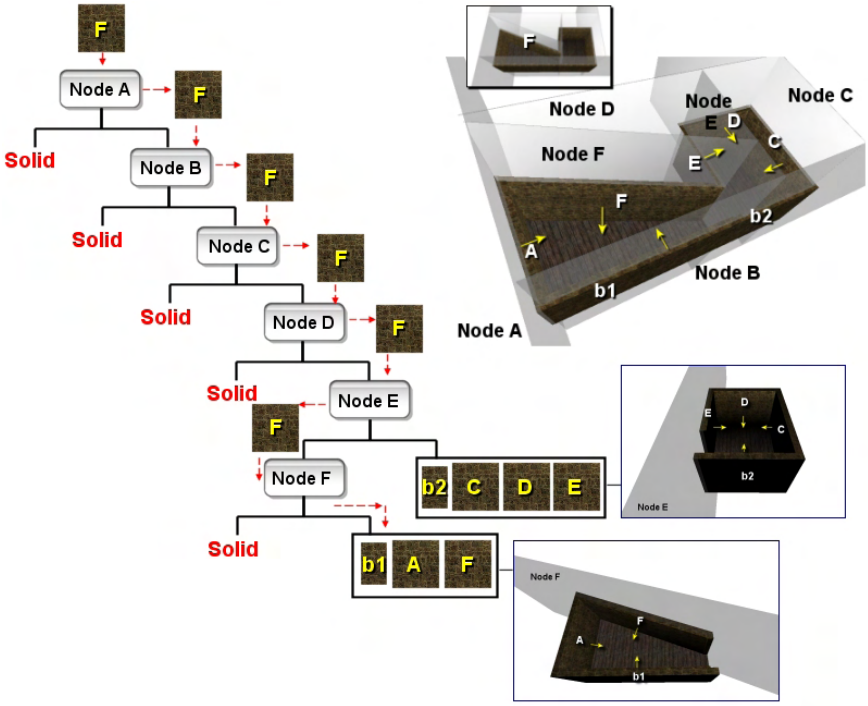


Figure 16.88

At node F, polygon F is clearly going to be found to be co-planar and same facing with the node since this was the polygon that was used to create this node. As we know, a polygon is always passed down the front of a node whose plane it was used to create, ensuring that the polygon ends up in an empty leaf which the node plane normal is facing into. We can see that this is the case in Figure 16.88. At node F, polygon F is passed down the front where it is added to the leaf that exists there. This leaf contains polygons A, b1, and now F, forming the second convex area that the non-convex geometry has been broken into.

Figure 16.88 shows our final tree which has been compiled such that if ever we traverse into a back leaf, we know we have reached solid space.

Figure 16.89 depicts our compiled tree along with three positions labeled A, B, and C. We can see in the top right image that points A and B are in solid space because they lie behind the polygons and have no line of sight with the front of any polygons in the level. Point C is in empty space as it is contained within the rightmost populated leaf. The tree diagram demonstrates what happens when we drop these three points through the tree and track their progress as they are classified against each node plane and sent down the front or back of each node depending on the classification result. As we can see, the tree gives us the correct result and verifies not only that two of these points are in solid space and one of them is in empty space, but it also tells us exactly in which solid and empty regions of the scene these points reside.

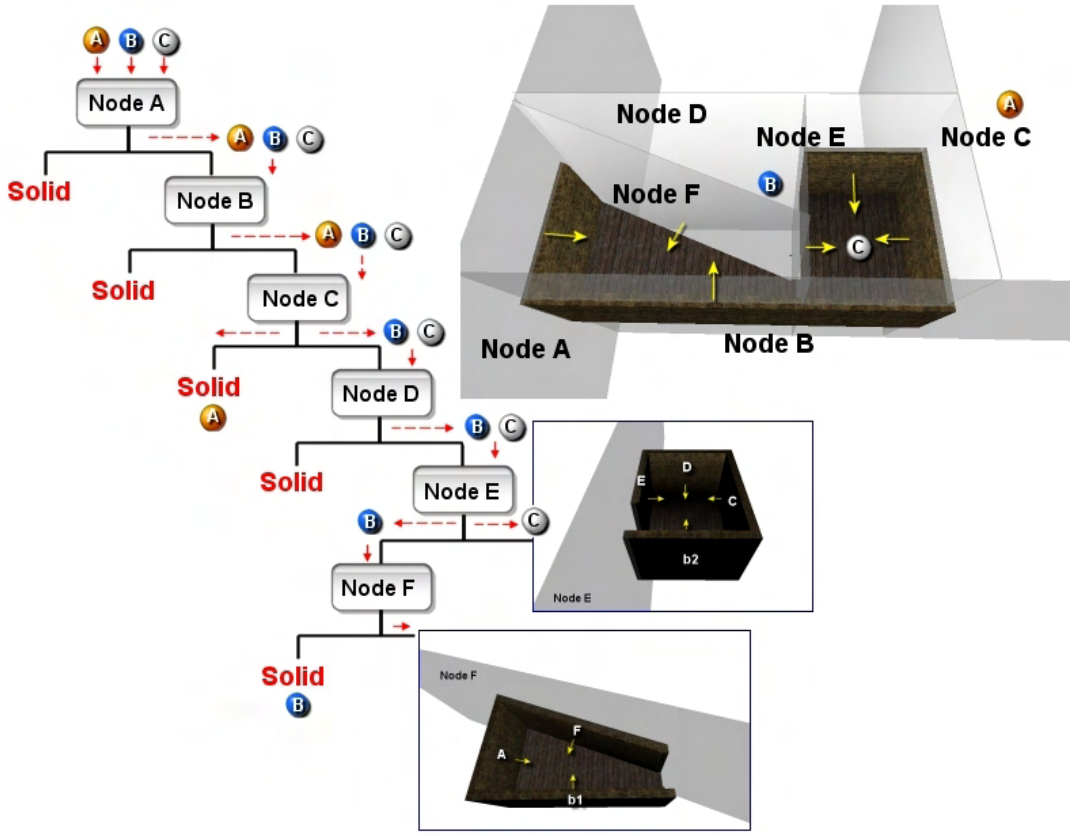


Figure 16.89

You should now be able to understand the above diagram and understand the reasons for each point's journey through the tree. As the tree shows, there are only two empty regions in this scene in which the player would normally be allowed to be located -- the empty leaf bounded by inward facing polygons b2, C, D and E (down the front of node E) and the empty leaf in front of node F bounded by inward facing polygons A, b1, and F. We can think of the polygons as providing the barrier between empty space and solid space and the means by which we stop one flooding into the other. If illegal geometry is supplied to the compiler that is exactly what will happen. We will end up with areas of space that are ambiguous because they exhibit traits that would classify them as being both solid and empty space. We

will look at some examples of illegal geometry in a short while and examine how it corrupts the solid/empty integrity of the tree.

16.3.4 Building the BSP Solid Leaf Tree Compiler

In our discussion so far we have discussed (for educational purposes) compilation of a leaf tree in two different stages. In the first stage we used the input polygon list to build the tree of nodes in the exact the same way we did when constructing our BSP node tree earlier in the lesson. However, we have assumed that once the polygon has been selected for a node it is discarded from the list. At the end of the first stage, we will have built an empty BSP tree (i.e., a BSP tree which contains no polygon data in its leaves). We then looked at how we could collect these polygons at the leaves during a second pass, where each polygon is sent down the tree and clipped to the nodes. The polygons eventually pop out at leaf nodes and these are the nodes to which the polygons are assigned. Of course, there is no reason to do this all in two separate passes as doing so would be inefficient. When constructing the tree, we have the polygon data at our disposal, so there is no reason why we cannot pass the polygons down the tree, clip them to the nodes, and collect them at the leaf nodes during construction. This allows us to implement the entire tree building process along with its static data storage in a single pass. This is how a BSP compiler usually operates and is how the BSP compiler that we will create in Lab Project 16.2 will operate.

Collecting the polygons at the leaf nodes during the build process has much in common with the way we constructed our quad-tree, oct-tree and kD-tree in the previous lessons. When constructing such trees, we generated an axis aligned split plane (or multiple split planes) at each node and divided the polygon list passed into that node into 2, 4, or 8 sub-lists depending on the type of tree being constructed. In the case of the kD-tree for example, at each node all of the polygons would be divided into two lists to be passed into the front and back child nodes. This same process continued until we reached a terminal node and all of the polygons that make it into that node were stored in a leaf structure that was attached to that node.

The same is going to be true with the BSP tree, although we have other things to consider, since the polygon list that makes it into each node is also being used to select split planes. In a nutshell, once a polygon has been selected as a splitter and a node plane created from it, it must still be passed down the tree, classified and potentially split against the remaining nodes in the tree (i.e., what was our ‘second pass’ earlier). The polygon that was used to create the root node for example, might be passed down the tree and split into multiple fragments, each assigned to a different leaf node. This is no different from how a polygon is passed down the kD-tree and repeatedly clipped to the nodes of the tree resulting in multiple fragments during the build process. However, what we must be mindful of is that once a polygon has been used as a split plane, it must never be selected as a splitter further down the tree by another node. This was not an issue in the node tree because as soon a polygon was selected to create a node plane, it was removed from the list and stored in the node. However, because we are continuing to pass the polygon data down the tree, we must make sure that it is not selected again. Furthermore, we must also make sure that if a polygon has been used as a node plane and later on down the tree it gets split by another node plane into two new child fragments, those child fragments must also not be used as splitters. This is obviously not very difficult to achieve; we can simply add a Boolean member to our

CPolygon structure called ‘BeenUsedAsSplitter’ (for example) which is set to true once the polygon has been used as a node plane. When this is passed down the tree into child nodes, each child node, when selecting a polygon from its input list to use as a split plane, will ignore any polygons in the list that have their ‘BeenUsedAsSplitter’ Booleans already set to true. This will prevent the polygon from ever being used as a split plane again until it eventually pops out in an empty leaf and is stored. If the polygon gets split further down the tree, we can simply copy the ‘BeenUsedAsSplitter’ status of the parent polygon into the two new child polygons so that they too will not be used as splitters as they continue their route down the tree. In fact, we must make sure that is the case for *all* co-planar same-facing polygons. That is, if a polygon is selected as a node plane, we must flag any polygons in the list that are co-planar and same-facing as having been used as a splitter too, so that they are not used to create additional planes down the tree. Since all co-planar polygons would generate the same split plane, it would be redundant to do otherwise.

In our final BSP compilation example we will bring these processes together to show how a piece of legal geometry can be compiled into a solid BSP leaf tree in a single pass.

In the leftmost image in Figure 16.90 we see a top down view of a simple scene (a room containing two triangular pillars). Once again, back face culling has been disabled so that we can see all the faces in the level and the yellow arrows depict the normals of each polygon and the node planes that they will create. The four outer walls form the walls of the room itself. These are facing into the room and as such, space behind these walls is considered solid space. In the center top section of the room there are two triangular constructs that are supposed to be simple pillars. These polygons all face out from the center of the pillar into the empty space of the room and as such, the areas bounded by the back faces of these pillars is naturally considered solid space. In the leftmost image we have placed a red ‘S’ to depict where solid space is assumed to be. In the rightmost image we show what this room might look like were the camera to be placed inside and we can see a portion of the two solid pillars at the far end of the room. A floor polygon has also been added in the rightmost image to give a better idea of what the room would ideally look like. However, to simplify the number of nodes we have to draw in our diagrams, we will ignore the floor and ceiling polygons and compile only the walls.

Figure 16.90

Each polygon in the leftmost image is labeled A through H, which will use to demonstrate the order in which each polygon will be selected to create a node plane at each step during the compilation process. As we have discussed, the polygons can be selected in any order and a valid tree will still be created. Of course, the number of leaves and polygons that result in the tree will be different, but the solid and

empty space will be retained and each leaf will represent a convex area described by the parent planes that intersect to form the boundary of that region.

Before looking at the tree diagram and following the compilation process step-by-step, Figure 16.91 demonstrates how the scene will be carved if the nodes are selected in the above order. It also shows where the empty leaves will be located, and can be used as a reference when we discuss the tree diagram that follows it.

Figure 16.92 depicts every step of compiling this level into a solid BSP tree and collecting the polygon data at the leaf nodes in a single phase.

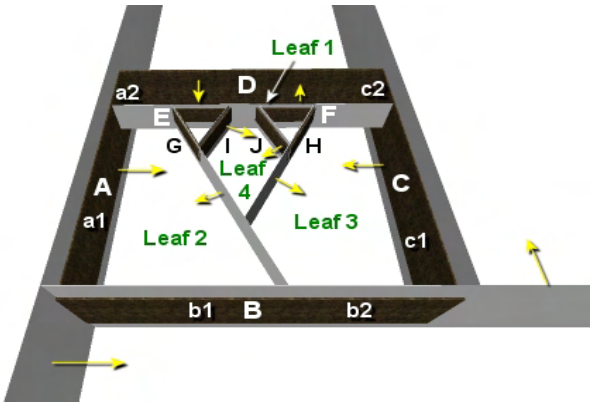


Figure 16.91

The polygon list passed into to the compiler contains polygons A through J. Polygon A is selected as the splitter first and its plane stored in the root node. Polygon A (along with any co-planar polygons) is marked as having been used as a splitter. Polygons that have already been used as splitters are highlighted red in Figure 16.92.

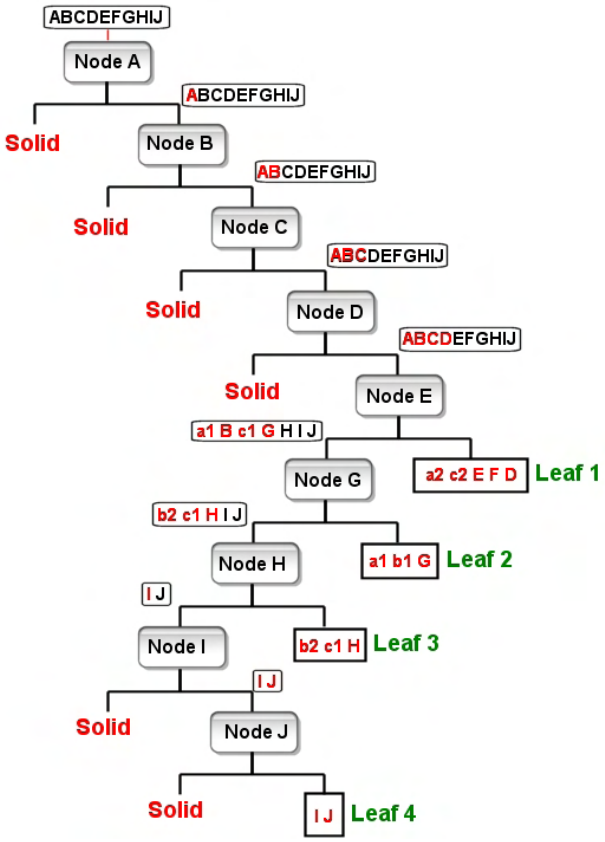


Figure 16.92

Every polygon classified against node A is found to exist in its front space, added to the front list, and passed down the front of the node. As no polygons made it into the back list of node A an empty leaf is created there, which we know will represent solid space. At the front child of A, a splitter is selected so a choice is made from polygons B through J. Polygon A is ignored by the splitter selection process as it has already been used. Polygon B is selected in this example and is marked as having been used as a splitter. All the polygons (**including A**) are classified against node B and are added to the front list. No polygons exist in the back list of node B so an empty (solid) leaf is created there. The polygons in the front list prompt a new child node to be created down the front of B and this time the selection for a splitter ignored polygons A and B as they have already been used. This process continues down to node E where we can see at this point, nodes A,B,C,D and E have been used as splitters. At node E, polygons A through J are classified against the node plane and two lists are created. Furthermore, polygons A and C span node plane E and are therefore split into polygons a1,a2 and c1,c2 respectively. The back list of node E contains polygons a1, B, c1, G, H, I and J and the front list contains polygons a2, c2, E, F, and D. Notice however that because polygons A and C have already

been used as splitters before they were split, the status of their parents is carried over into the children. Polygons a1, a2, c1 and c2 are all highlighted red, indicating they have already been used as splitters and should not be selected again. We can see in Figure 16.92 that when the front list is passed down the front of node E, every polygon in this list has already been used as a splitter and therefore, we have no more nodes to create down this side of the tree. When this is the case we know it is time to create a leaf and that every polygon in this list must lay on the boundary of a convex area. These polygons are added to Leaf 1 which is attached to the front of node E.

The back list of node E contains polygons a1, B, c1, G, H, I and J. These have not yet all been used as splitters so a new child node still needs to be created down the back this node. From this list, only polygons G, H, I, and J are considered as node plane candidates as they are the only ones that have not yet been used as splitters. In this example, node G is chosen next which splits polygon B into b1 and b2 as shown in Figure 16.94. When all the polygons are classified against G we end up with a front list containing polygons a1, b1 and G and a back list containing polygons b2, c1, H, I and J. As all the polygons in the front list have been used as splitters a new leaf is created down the front of node G and polygons a1, b1 and G are added to it.

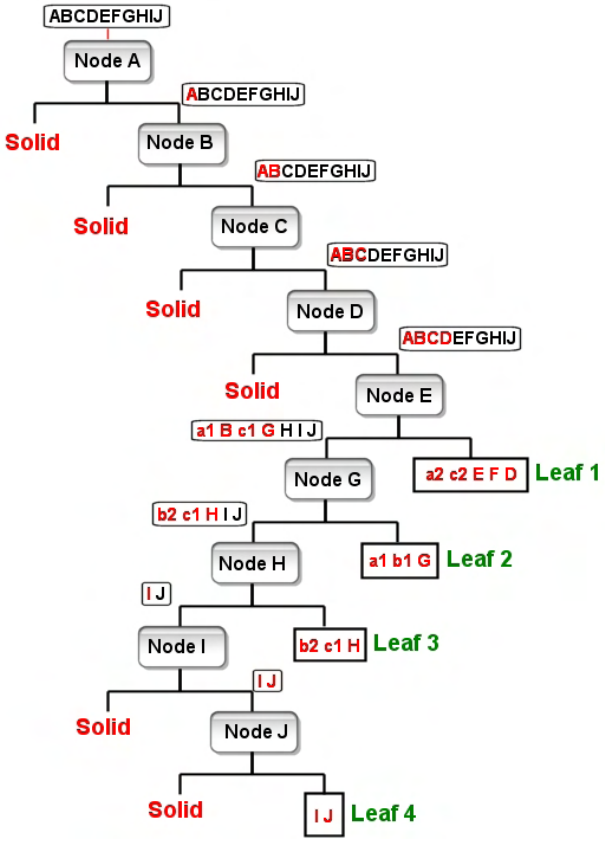


Figure 16.93

In the back list of node G we have three polygons which have not yet been used as splitters: H, I and J. Polygon H is selected next. When the remaining polygons (b2, c1, H, I and J) are classified against node H we end up with a front list containing polygons b2, c1, and H which have all been used as splitters. This means that these polygons are added to a new leaf (leaf 3 in the diagram) which is attached to the front of node H.

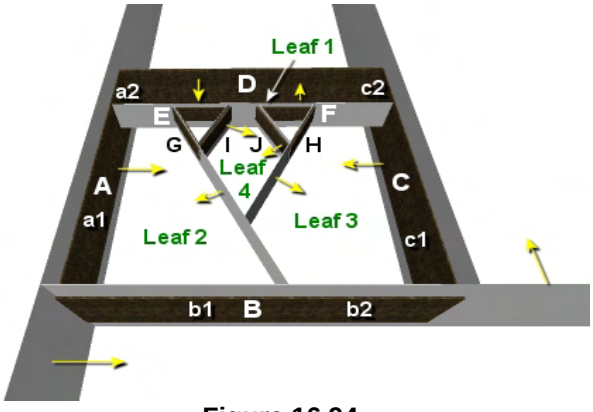


Figure 16.94

The back list of node H contains polygons I and J which have not yet been used as splitters. Polygon I is selected next to create node I. Both of these polygons exist in the frontspace of node I, so a solid leaf is created down the back of node I. The polygon list passed into the front of node I contains polygons I and J and only J has not yet been used as a splitter. Thus J is selected next and marked as having been used. These polygons are then classified against node J and both found to exist in the front halfspace. This means the creation of a solid leaf behind node J. As the polygons in the front list have now all been used as splitters we have no

more nodes to create and polygons I and J are added to a new empty leaf which is attached to the front of node J.

We have now just compiled the complete solid leaf BSP tree on paper. Before moving on, study Figures 16.93 and 16.94 and make sure you fully understand how the tree was constructed and where the solid and empty areas are. Notice how the polygons contained in the leaves in Figure 16.94 marry up with the polygons that are shown in the front leaves in Figure 16.93. Notice as well that empty space is always the region of space that has the polygon normals facing into it and how solid space is always separated from empty space by the backs of polygon along its boundaries.

Let us now examine some placeholder code that could be used to compile a BSP tree. For the exact code, please consult the accompanying workbook and lab project. The code shown here is just to demonstrate the basic concepts and logic that must be employed in a solid leaf BSP compiler.

This demonstration code assumes that a class exists called CBSPTree and that we are showing only the details of the compilation functions. It is assumed that before the CBSPTree::CompileTree function is called by the application, all static polygon data that should be compiled into the tree has been registered with the tree and added to the tree's linked list of CBSPPolygon structures pointed at by the member variable m_pFaceList. The CBSPPolygon structure is assumed to have a 'UsedAsSplitter' member that will be set to false for all input polygons prior to the commencement of the compilation process. As discussed, this Boolean member is set to true when a polygon is used as a node plane or if it is found to be co-planar and same facing as a node plane.

Finally, it should be noted that just because the leaves of the BSP tree (and the nodes) no longer represent regions that can tightly fit into an axis aligned bounding box, we can still store AABBs at each node and use the same hierarchal bounding box tests for hierarchical visibility determination and collision querying as implemented in the previous lesson. It should also be noted that because the leaves are no longer box shaped volumes, the bounding box we store at the node will no longer tightly bound the leaf in the typical case.

Imagine for example a leaf that was shaped like a view frustum (see Figure 16.95). We can see that the bounding box that was compiled for the leaf would be a loose fit. It is possible during a collision query for example, that the collision volume would intersect the bottom near corner of the leaf box (the minimum world extents corner of the leaf's bounding box). In such a case the leaf itself would not be inside the query volume (only its box would) but it would return true for a collision. In such a case, the polygons in the leaf would be tested needlessly for collision. However, this is certainly preferable to testing all the polygons in all the leaves and fits in with the visibility system and the collision querying system we have used for the other tree types.

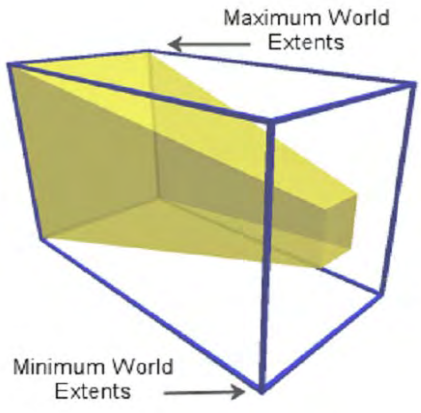


Figure 16.95

The top level function in this example is called CompileTree which simply creates the root node and invokes the recursive compilation process by calling the BuildBSPTree function. This function is passed the newly allocated root node and a pointer to the head of the polygon linked list.

```
HRESULT CBSPTree::CompileTree( )
{
    // Validate values
    if (!m_pFaceList) return BCERR_INVALIDPARAMS;

    m_pRootNode = new CBSPNode;

    // Compile the BSP Tree
    BuildBSPTree( &m_pRootNode, m_pFaceList));

    // Success
    return BC_OK;
}
```

The BuildBSP tree function has the task of populating the node with a separating plane that will be chosen from the list of polygons passed into the node. Only polygons in the list that have not yet been used as a splitter will be considered as a separating plane for this node.

```
HRESULT CBSPTree::BuildBSPTree( CBSPNode *pNode, CBSPPolygon * pFaceList )
{

    CBSPPolygon     *TestFace  = NULL, *NextFace      = NULL;
    CBSPPolygon     *FrontList = NULL, *BackList      = NULL;
    CBSPPolygon     *FrontSplit = NULL, *BackSplit    = NULL;
    CBSPPolygon     *Splitter   = NULL;
    CLASSIFY      Result;
    int           v;

    // Select the best splitter from the list of faces passed
    Splitter = SelectBestSplitter( pFaceList,
                                   DEF_SplitterSample,
                                   DEF_SplitHeuristic);

    if (!Splitter) { Error has occurred so do cleanup here }
```

We saw earlier in the node tree discussion that the SelectBestSplitter function is passed a list of polygons from which it will select one that should be used to create the node plane for the current node. The final two parameters to this function control how many polygons should be sampled in this test and the ‘splits versus balance’ heuristic that should be used when selecting the polygon. In the above code, the final two parameters to this function are assumed to be variables or #defines that you would like used for these values. The SelectBestSplitter function will have been slightly modified from the node tree version in that it must now ignore any polygons in the passed list which have previously been used as splitters (i.e., have their ‘UsedAsSplitter’ Booleans set to true). However, they will still be classified against each potential splitter candidate and contribute to the scoring of the balance versus splits heuristic. After all, just because a polygon has been used as a split plane does not mean that it will not get split into multiple fragments by node planes selected further down the tree during the building process. We will not show the modified code here to the SelectBestSplitter function as the full source

code will be discussed in the accompanying workbook. However, just know that all we have done is modified the outer polygon loop so that any polygons that have been used as splitters are not considered for a new split plane.

The function returns a polygon. In this example code we will assume that each CBSPPolygon also stores the polygon's plane and as such we can simply fetch the plane from the polygon and store it in the current node being visited, as shown below. We also set the returned polygon's (Splitter) UsedAsSplitter Boolean to true so it, or any fragments it gets split into, will never be used to create another node.

```
    // Flag as used, and store plane
    Splitter->UsedAsSplitter = true;
    pNode->Plane = Splitter->Plane;
```

Now that we have the separating plane stored at the node our next task is to loop through every polygon in the list and classify it against this plane and add it to the front or back list for this node depending on the classification result. Any polygons that span this plane will be replaced by two split fragments that exist in each halfspace and will be added to their respective lists.

In the following code we first set up a loop to iterate through each polygon in the linked list. The polygon we are current testing against the node will be stored in the local variable 'TestFace'. We also store its plane in 'Plane' for easy access. Notice that (just as we did in the node tree) we also store a temporary pointer to the next polygon in the list so that we still have access to the next polygon to be processed even when the current polygon being processed is removed from the list. This is important as currently the only access we have to the next polygon in the list is via the current polygon's next pointer (TestFace->Next). If TestFace gets deleted from the list during a split operation or a leaf assignment, we will still need to access the next polygon.

```
    // Classify faces
    for (  TestFace = pFaceList;
           TestFace != NULL; TestFace = NextFace, pFaceList = NextFace )
    {
        // Store plane for easy access
        CPlane Plane = TestFace->Plane;

        // Store next face, as 'TestFace' may be modified / deleted
        NextFace = TestFace->Next;
```

Now it is time to classify the current polygon (TestFace) against the node plane to find out whether it is in the front or back halfspace, spanning, or on-plane with the node. If the current polygon's plane is the same as the plane of the polygon that was selected as a splitter, then we know we have a polygon that is on-plane. Otherwise, we classify the polygon against the plane as we normally do by using a function called ClassifyPolygon which accepts the plane we wish to classify against (which in this case is the plane that was used as the node plane) and the vertices of the current polygon being tested.

```
        // Classify the polygon
        if ( TestFace->Plane == Splitter->Plane )
        {
            Result = CP_ONPLANE;
```

```
        } // End if uses same plane
        else
        {
            Result = ClassifyPolygon( SplitterPlane ,
                                      TestFace->Vertices,
                                      TestFace->VertexCount,
                                      sizeof(CVertex));

        } // End if differing plane
```

Now we have the classification result of TestFace stored in the Result local variable. This will contain CP_ONPLANE, CP_SPANNING, CP_FRONT, or CP_BACK. Next we will enter a switch statement and take the appropriate action in each case.

If the result is CP_ONPLANE then we know that the vertices of the test polygon lay on the node plane but we do not know yet whether this polygon faces into the same frontspace as the node or has an opposing normal. As discussed several times throughout this lesson, this is important because if the normal faces in the opposite direction, it should be added to the back list. Otherwise, it should be added to the front list. In the next section of code we assume the implementation of a function called ‘SameFacing’ which compares the two normals passed in as parameters and returns true if they are facing into the same halfspace.

```
        // Classify the polygon against the selected plane
        switch ( Result )
        {
            case CP_ONPLANE:

                // Test the direction of the face against the plane.
                if ( SameFacing( Splitter->Plane.Normal , pPlane->Normal ) )
                {
                    // Mark matching planes as used
                    if (!TestFace->UsedAsSplitter)
                    {
                        TestFace->UsedAsSplitter = true;

                    } // End if !UsedAsSplitter

                    TestFace->Next  = FrontList;
                    FrontList       = TestFace;
                }
                else
                {
                    TestFace->Next  = BackList;
                    BackList        = TestFace;
                } // End if Plane Facing
                break;
```

As you can see, we pass in the node plane normal and the normal of the test polygon and if found to be equal we know we have a co-planar same-facing polygon. When this is the case we set its

UsedAsSplitter Boolean to true since using this polygon as a splitter further down the tree would be redundant as the scene has already been divided by the same plane. We also add the polygon to the front list by assigning the test face's next pointer to point at the current head of the list and then reassign the pointer to the head of the list to point at the test face. Essentially, we are just adding the polygon to the head of the front list. Notice in the above code however that if the normals are not facing in the same direction in the on-plane case, the polygon is added to the back list and its UsedAsSplitter status is unaltered.

The CP_FRONT and CP_BACK cases are delightfully simple. If the polygon is located entirely in the frontspace of the node then we just add its pointer to the front list of polygons being compiled. If the polygon is located in the backspace of the current node then we add it to the head of the back list currently being compiled for this node.

```
            case CP_FRONT:
                      // Pass the face straight down the front list.
                      TestFace->Next            = FrontList;
                      FrontList                 = TestFace;
                      break;

            case CP_BACK:
                      // Pass the face straight down the back list.
                      TestFace->Next            = BackList;
                      BackList                  = TestFace;
                      break;
```

When a polygon is found to be spanning the node it must be split into two child fragments just as was the case with the node tree. However, after we have created the two new child fragments, we must set their 'UsedAsSplitter' Booleans equal to the parent polygon status prior to the parent polygon being deleted. This will make sure that if the parent polygon has already been used as a splitter, this state is inherited by the children so that they are not selected as splitters later. As discussed, this would be redundant and would create many unnecessary nodes. Shown below is the CP_SPANNING case and the final section of the polygon classification loop.

```
            case CP_SPANNING:

                // Allocate new front fragment
                FrontSplit          = CBSPPolygon;
                FrontSplit->Next    = Frontlist;
                FrontList           = FrontSplit;

                // Allocate new back fragment
                BackSplit           = new CBSPPolygn;
                BackSplit->Next     = BackList;
                BackList            = BackSplit;

                // Split the polygon
                TestFace->Split( Splitter->Plane, FrontSplit, BackSplit);

                // Copy over status of parent into children
                FrontSplit->UsedAsSplitter = TestFace->UsedAsSplitter;
                BackSplit->UsedAsSplitter  = TestFace->UsedAsSplitter;
```

```
                // Remove original polygon
                delete TestFace;

                      break;

            } // End Switch

      } // End while loop
```

As you can see in the above code, two new polygons are created, FrontSplit and BackSplit. These are fed into the parent polygon's Split method along with the node plane. When the split function returns, FrontSplit will contain the polygon fragment that exists in the frontspace of the node plane and BackSplit will contain the fragment that exists in the backspace. Before deleting the parent polygon, we copy the value of its UsedAsSplitter Boolean into each of the children.

When the while loop shown above exists, we will have compiled a front list and a back list of polygons describing exactly which polygons should be passed down the front and back tree of the current node.

The first thing we do is test to see if the back list has any polygons in it and whether any of them have still not yet been selected as splitters. If there are still polygons in the back list which have not been selected then all is well and we know we have to create a new back child node and keep on creating nodes from this list. If the back list has a list of polygons which have all been used as splitters then we have a real problem. Under normal circumstances (were we compiling a basic leaf tree) we would just store the polygons in a back leaf, and we could certainly do that. However, we have also discussed that if we have been passed legal geometry, this situation should never occur. That is, back leaves will always be empty and we should never be in a situation where we have a list of polygons that have all been used as splitters existing at the back of a node. When this happens, we have been given illegal geometry and we run the very real risk that the solid/empty information will be corrupt. Now, normally if the artist has created the level without regard to the technology being used, then the level will be highly illegal and there is not much you can do except modify the source level to get rid of the offending geometry (we will discuss such techniques in a moment). However, it is sometimes the case that the geometry was defined in such a way that it is technically legal but due to floating point accumulation errors or perhaps slightly sloppy object placement by the artist, a polygon's normal may face into a solid leaf. When this is the case, it often means this small sliver of polygon which has ended up in a back leaf can simply be deleted as it exists within the solid space of another object. This is the approach we take in this example. If any polygons end in back leaves, we will assume it is a floating point accumulation error and simply delete the offending fragments. However, while this works very well for compiling geometry that is essentially legal but with a few minor issues, geometry that has been assembled without any concern for solid/empty legality cannot be fixed by this simple solution. If the geometry is extremely illegal then this method could well delete half the level.

> **Note:** It is important that your project artist be aware of the rendering technology you are using and design the artwork in a compliant manner. Although we will discuss techniques for correcting illegal geometry in the next section, as well as gain a greater understanding as to what causes it, the artist must take responsibility for developing assets that can be used by the rendering solution you ultimately decide to employ.

In the following code we remove any illegal fragments that end up facing into back leaves. This assumes the implementation of a function called ‘DeletePolyList’ that, when passed the back list, will delete all the polygons contained within.

```
     // If No potential splitters remain, free the back list
     if ( BackList && CountSplitters( BackList ) == 0 )
     {
        // Remove illegal faces
        DeletePolyList (BackList);
        BackList = NULL;

     } // End if No Splitters
```

We can really not stress enough that the above section of code is absolutely no substitute for correct solid/empty level design. It fixes problems in situations where floating point inaccuracies introduced by the myriad of clipping operations that can be done on a polygon during the compile process cause its vertices to drift off the original plane and end up in solid space. But it is not a magic wand that can be thrown at any polygon soup with the expectation of compiling a perfect tree.

Now that we have the front and back lists of the current node compiled, we will construct a bounding box to be stored at that node that will bound all the polygons that exist down its front and back side. In the following code it is assumed that the CBSPNode structure has a function called CalculateBounds that, when passed two lists of polygons, will construct its bounding box to be large enough to contain the vertices of all polygons stored in those lists. Although this bounding box will not fit the geometry as tightly as the nodes of a quad-tree or an oct-tree (where the polygons have been clipped into box space regions), they can still be used to coarse cull entire branches of the tree during visibility and collision queries.

```
    // Calculate the nodes bounding box
    pNode->CalculateBounds( FrontList, BackList );
```

In the next step we test to see if any of the polygons in the front list are yet to be selected as splitters. If this is not the case then all the polygons in the front list have already been selected as splitters further up the tree and we have no more nodes to select. We also know that the polygons in this list will lie on the boundary planes of a convex region of empty space (an empty leaf). The polygons will also be facing into this empty leaf so our task is simple: create a new leaf, add the polygons to it, and attach it to the node’s leaf member. If there are still polygons in the front list that have not yet been selected as splitters then more subdivision must occur down the front of this node. When this is the case we create a new node, attach it to the current node’s Front pointer and recur into that node with the front list.

```
    // If all splitters are used in frnt list create front leaf
    if ( CountSplitters( FrontList ) == 0 )
    {
        // Add a new leaf and store the resulting faces
        CBSPLeaf * FrontLeaf = new CBSPLeaf;
        pNode->Leaf = FrontLeaf;
        CBSPLeaf->AddPolygons ( FrontList );
    }
    else
```

```
    {
        // Allocate a new node and step into it
        CBSPNode *pFrontNode  = new CBSPNode;
        pNode->Front          = pFrontNode;
        BuildBSPTree( pNode->Front, FrontList );

    } // End If FrontList

    // Front list has been passed off, we no longer own these
    FrontList = NULL;
```

When we reach the bottom of the above code, all children down the front of the node will have been created and all the polygons in this node's front list will have had their pointers stored in leaves. This means we should not delete the polygons stored in the front list since their pointers have been copied into other locations; we can simply set the front list pointer to NULL.

What may seem strange in the above code is that we assign the front leaf to a node member called 'Leaf'. As a node can have both front and back leaves, should this not be called 'FrontLeaf' instead? Well, we could, but since we know for a fact that our BSP tree will never store polygon data in back leaves, why bother creating empty leaf structures and attaching them to nodes when nothing will ever be stored in them? Instead, we will simply make the node's Back pointer, which normally points to a back child node, dual purpose. If this member is not NULL then it means there is a child node attached to the back of the current node. If this pointer is NULL, it must mean a leaf exists down the back of this node. However, all we need to know is that a solid space leaf is represented by a NULL pointer down the back of a node and we have all the information we need. Therefore, the only leaves we ever actually have to create and store information for are front leaves, which will be pointed to by the node's Leaf member. If traversing the tree we find that we need to traverse down the back of a node that has its back node pointer set to NULL, we know that we have traversed into solid space. Here is the following and final section of the compilation function.

```
    // If the back list is empty flag this as solid
    // otherwise push the back list down the back of this node
    if ( !BackList )
    {
        // Set the back as a solid leaf
        pNode->Back = BSP_SOLID_LEAF;
    }
    else
    {
            // Allocate a new node and step into it
            CBSPNode *pBackNode = new CBSPNode;
            pNode->Back = pBackNode;
            BuildBSPTree( pNode->Back, BackList);

     } // End If BackList

    // Back list has been passed off, we no longer own these
    BackList = NULL;


    // Success
    return BC_OK;
```

```
}
```

As you can see, if there are no polygons in the back list we know there must be a solid leaf behind this node so we set the back pointer to NULL. Otherwise, we need to further subdivide the space behind this node and we create a new node, attach it to the current node's Back pointer, and recur into the back child with the back list.

We have just examined everything involved in writing a solid leaf BSP compiler. This code will generate a BSP leaf tree which can be traversed like any other tree to perform visibility and collision queries. The fact that we have stored bounding boxes at each node also allows us to use the same AABB hierarchical queries on the tree as before, although we may sometimes end up traversing into leaves where the geometry does not intersect the query volume. As discussed earlier, this is because the bounding box of a node will not be a tight fit around its geometry. Therefore, it is possible that a ray for example, may be determined to intersect the bounding box of a leaf (or node) even if the ray does not intersect the convex region of the leaf. Of course, this simply means that we may end up testing polygons in a leaf that are not actually contained in the query volume in some circumstances. The same frustum culling traversal can also be performed on the tree although once again, because of the loose fitting bounding boxes around the nodes, we may find that we traverse into nodes and render some leaves that are not actually contained within the frustum.

When we consider that the polygon aligned solid leaf BSP tree will on average create much larger trees than quad-trees or kD-trees for example, one might imagine why this tree would be favored over our previous tree types. To be clear, if all we were going to do is plug our solid leaf BSP tree into the leaf bin rendering system we developed in the previous lesson, this would be a valid point. The BSP tree would be outperformed by those other trees in the typical case. However, because the BSP tree contains the solid/empty information, it makes CSG operations possible (discussed later in this lesson) and allows for very efficient line of sight tests to be carried out on large scene databases. Furthermore, and by far the most important feature, is that it allows us to construct a potential visibility set for our level which will increase the performance of our current rendering systems by an order of magnitude (in the case of a highly occluded level). Therefore, in the next lesson we will see how the BSP tree, coupled with the potential visibility set, will be our rendering technology of choice going forward and one of the core technologies that is carried forward into Module III when we start to construct our game engine.

16.4 Illegal Geometry and Hidden Surface Removal

It should be apparent at this stage that the 'solid' aspect of a solid BSP leaf tree is entirely dependant upon the geometry that is built by the artist and ultimately fed into the compiler. If the artist generates scenery in which the various different areas of the scene are not properly closed or bounded, then our ability to determine the correct nature of the space described by each leaf will be lost. While the geometry that is compiled is still a very important factor in whether or not the integrity of the solid/empty information is retained, there are solutions we can employ to ensure that this situation is less of an issue. In this section we will discuss exactly what illegal geometry is and how it can be avoided during the asset development phase. Illegal geometry is caused by a condition that can be summed up as follows:

"When the front of one or more polygons (or some part of them) can see the back of another polygon, we have illegal geometry. This means that the boundaries between solid and empty space have leaked into one another, making solid/empty space determination impossible."

Figure 16.96 shows the simplest case of illegal geometry. We have two isolated polygons positioned such that the front of polygon can 'see' the back of polygon B. If we look at the tree diagram to the right we can see that the following happens:

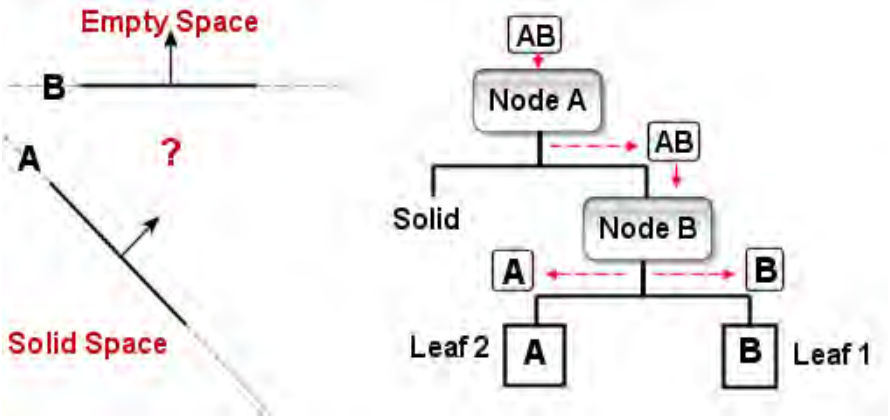


Figure 16.96

At the root node polygon A is selected as the splitter and when classified against this node plane, polygons A and B are sent down the front where B is selected as the next node. Behind node A no polygons exist in the back list so this is correctly identified as a solid leaf. Remember, solid space is the space behind the polygons. At node B we classify the polygons and find that polygon A is added to node B's back list and polygon B is added to the front list. As polygon B has already been used as a splitter this means that to the front of node B is a leaf containing polygon B. This space is correctly identified as empty space. Remember, empty space is the space located in front of the polygons. However, polygon A is passed down the back of node B but has been used as a splitter, so what should we do with it? We could delete it but that would make one of our polygons disappear and would describe the region of space between A and B as being solid space. This would be incorrect however as this space is in front of polygon A and therefore, should be empty space as it is the only space from which polygon A can be viewed. We could alternatively make this a back leaf and assign polygon B to it, but leaves with polygons assigned represent empty space therefore this would identify the region between polygons A and B as empty space. This is not correct either as in this space the viewer can see the back of polygon B, which would be removed during back face culling. The player should never be allowed inside regions of space where the backs of polygons can be seen. As you can see, this situation cannot be resolved and the region of space between polygons A and B cannot be added to any category. This is illegal geometry.

Of course, the above scenario is a little too simplistic as it is highly unlikely that any professional artist would ever construct a level which such glaring geometrical errors. However, there are ways to place geometry in a level that seem very sensible but which still cause illegal geometry, as shown next.

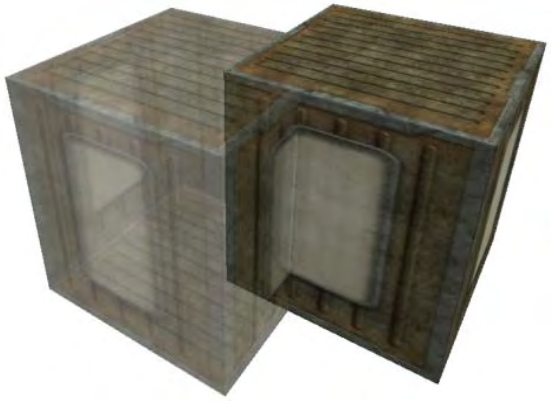


Figure 16.97

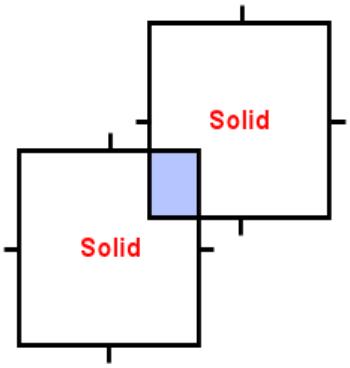


Figure 16.98

Imagine the scenario depicted in Figure 16.97 in which the artist has accidentally (or perhaps on purpose) overlapped two simple box shaped objects. The closest box has been rendered with alpha blending enabled so that we can more easily see that there are portions of the neighboring box intersecting its interior. Figure 16.98 shows the particular problem that our solid BSP leaf tree compiler would have in resolving this type of scenario during compilation. We know by experience that the space inside the two cubes should be defined as solid. However, due to the fact that some of the polygons intersect the interior space of each cube, we find a situation in which that same space which we know to be solid, exists in the front halfspace of one or more polygon fragments. That is, some portions of the polygons making up the boxes are front facing into the solid space of the other crate. This situation is one that we would commonly refer to as a *leak*. This is where solid and/or empty space – as defined by our tree and leaf layout – has 'leaked' into an area in which it should never have existed.

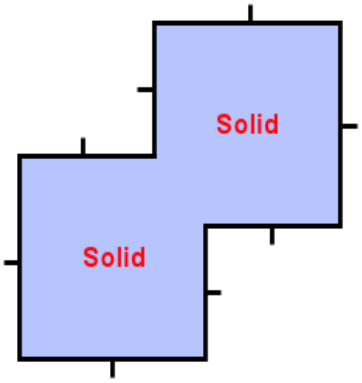


Figure 16.99

Figure 16.99 demonstrates the only possible arrangement of geometry in which this situation can be resolved. We can see here that the fragments of any polygons which are embedded in the solid space of either of the cubes have been removed. In doing, so we remove the ambiguity introduced by the erroneous front faces in solid space.

The process by which we go about removing these intersecting polygon fragments is called *hidden surface removal*. Put simply, this process is intended to remove any polygons that can never possibly be seen because they are contained within the solid space of other primitives in the scene. Not only can this remove a significant amount of overdraw in our application, it also takes care of the problematic situations outlined in this section. Something that you may have heard discussed in the past is the term 'Constructive Solid Geometry' (or 'Boolean Operations'). This term covers a multitude of topics, but in principal these techniques provide us with the means to perform various geometric operations on multiple objects. One example is the ability to merge the volumes of two primitives together in the cases where they might intersect.

In the next section we will see how we can use one of the techniques known as the 'union' operation to resolve many such types of problematic geometric situations prior to BSP compilation. This union operation is analogous to the hidden surface removal process depicted above and will help to ensure that we are able to create an accurate and very stable solid leaf BSP tree. Thus, even if the geometry does

contain such intersecting objects, the hidden surface removal techniques we will implement in this lesson will allow us to remove any polygon fragments that are found to reside in the solid space of another object. The GILES™ world editor can perform hidden surface removal on the entire level if you chose to select all brushes and then perform the union CSG operation on them. This will merge all the meshes in the scene into a single mesh, removing any polygon fragments that lie in solid space. However, if you are developing your levels in a package like 3D Studio MAX™, which does not employ the same HSR-oriented world building techniques, you may find that you create levels with many such hidden surfaces that would cause a BSP compiler to fail. Therefore, the BSP compiler we create in Lab Project 16.2 will also have an HSR processor that can be invoked as a pre-compile step to remove all such hidden surfaces and merge all meshes contained in the file into a single mesh which is then fed into the BSP compiler.

Although HSR techniques can remove hidden surfaces from within the solid space of an object, the individual meshes/brushes comprising the scene must be constructed from legal geometry. As long as all of the individual meshes comprising the scene can themselves be compiled into a solid leaf tree individually, we can perform a union operation to merge all these meshes into a single mesh with all hidden surfaces removed. However, if the individual components making up the scene contain intersecting/hidden surfaces, HSR will fail to fix the problem and the only cure is for the artist to fix this geometry manually.

Before we move into the final section of this lesson and discuss constructive solid geometry (a superset of hidden surface removal) we will look at some examples of legal geometry so that we are clear exactly how easy it is to create. Experience has shown that virtually every problem that a student has getting the BSP tree to operate and query correctly comes down to it being fed illegal geometry, and a misunderstanding of the various situations that cause illegal geometry during object placement in a level. By looking at some common examples of illegal geometry and how they can often be cured with the union CSG operation supported by many world editors and modelling packages, we will get a good feel for what to do and what not to do during the level building process.

Figure 16.100 depicts a scene created in the GILES™ world editor. It is a very simple level comprised of a room containing some pillars. It all looks pretty inoffensive until we examine how the level was created. The room itself was constructed by hollowing out a cube so that in the center of the cube we can paint the inward facing walls, floor and ceiling polygons with textures. All is well so far; the solid space of the room is contained between the polygons that form the interior and exterior sides of each wall. The outward facing polygons of the room can obviously not be seen in this image as we would have to walk outside for this to be the case, but it does not matter for this discussion. The room is perfectly valid geometry. It is a single mesh that respects the solid and empty space rules of construction.

What makes this geometry illegal is the placement of the four pillar meshes. Each pillar is constructed from three meshes: two square blocks that press against the floor and ceiling and a central cylindrical column. The cylinder is also resting up against the blocks. The blocks and the cylinder columns are closed meshes, so let us now see why this is a problem.

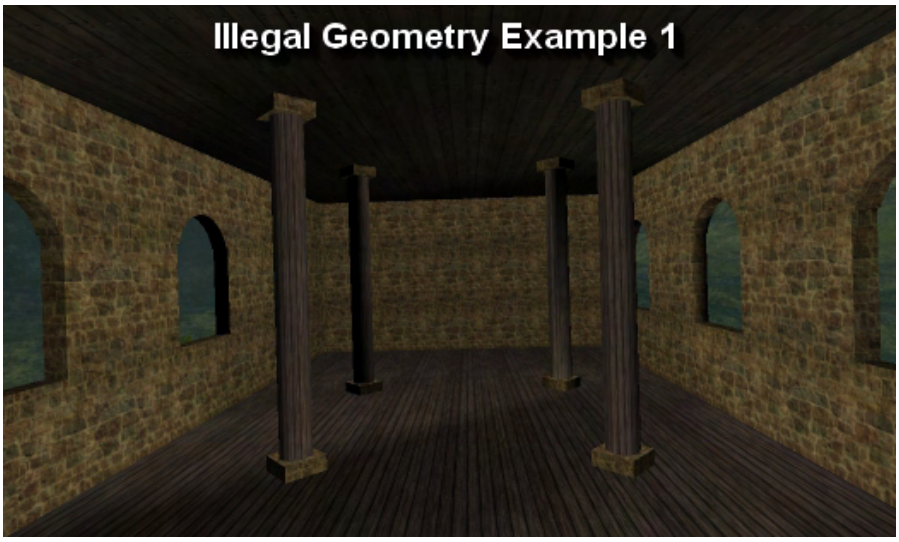


Figure 16.100

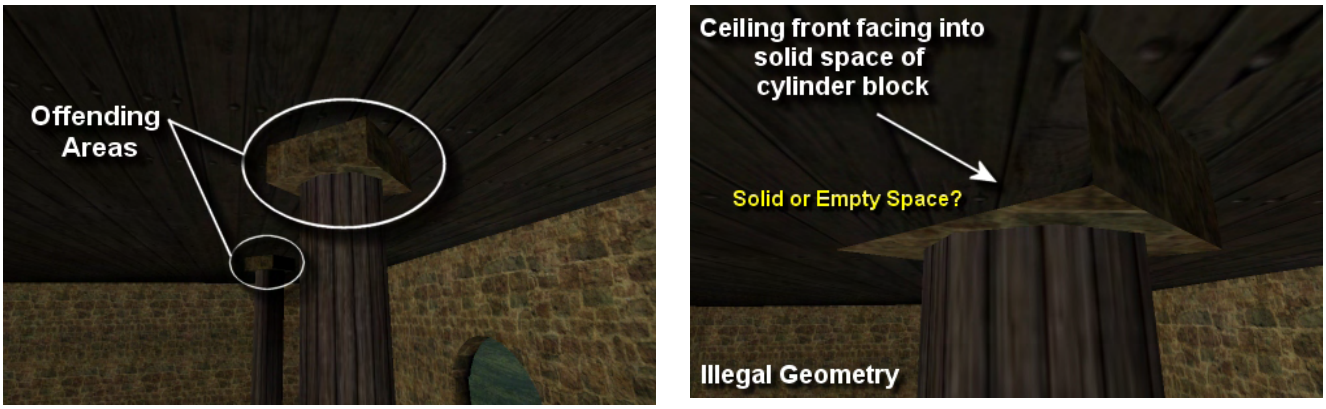


Figure 16.101

Figure 16.101 highlights the first problematic area of this scene. The pillar blocks themselves are pressed up against the ceiling, as shown in the rightmost image. As the top face of the pillar block and the roof polygons are co-planar, we have a situation where a section of the roof polygon is facing into the solid space of the pillar blocks. This is shown in the image on the right where we have removed a side face of the pillar block so that we can see what is happening inside. As you can see, the ceiling polygon is front facing into the solid space of the pillar block, which we know to be exactly what causes our solid BSP tree compiler to fail. Although it cannot be seen here, the top face of the pillar block would also be facing up into the solid space behind the ceiling polygons, so we have illegal geometry on two counts. Obviously, this will be the same situation for every cylinder block in the roof. The cylinder blocks at the bottom of each pillar will also have the same conflict with the floor polygon.

One very inelegant way to deal with this problem is to move the cylinder blocks ever so slightly away from the roof and floors so that empty space can flow between them. You may get away with this, but it certainly is not the answer we are looking for. Apart from being tedious for the artist to manually adjust everything like this, it may also happen that the user of our game will notice that the pillars are floating

in mid air. No, this definitely is not the way to go since this is a limitation we just can not live with. Luckily the hidden surface removal routines we will develop will come to the rescue by performing a union operation between the pillar block and the mesh of the room (which contains the roof polygon). The block will become part of the room mesh and the offending surfaces will be removed.

In Figure 16.102 we show the result of a union operation between the pillar block and the room mesh, which removes these hidden surfaces. As you can see, the section of the ceiling polygon that was contained inside (or co-planar) with the solid space of the pillar block has been carved away and although it cannot be seen here, so too is the top face of the pillar block that was pressed up against the ceiling.

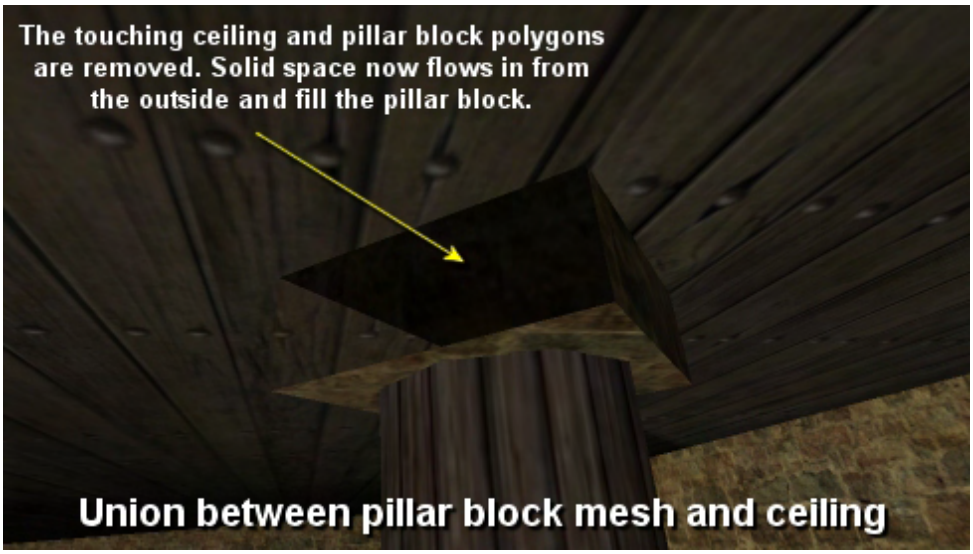


Figure 16.102

The pillar block and the room mesh have now been merged into a single mesh and the hidden surfaces have been removed. Solid space will now flow thorough this hole and fill up the inside of the block. Note that in this image we have removed the left face of the block so that we may observe what is happening inside. Obviously, if we were not to do this we would not notice any difference in the level. It is the *hidden surfaces* that have been removed by the union operation and since these polygons are hidden, we would not be able to visibly see that they have been removed.

Figure 16.103 illustrates where problems still exist. In this image, the front and back faces of the cylinder invisible so that we can see what is happening inside the solid space of the cylinder. As we can see, we have the same situation. This time, the bottom face of the block is co-planar with the top face of the cylinder (not seen here because of back face culling) which means the bottom face of the block is front facing into the solid space of the cylinder and vice versa. This problem can be resolved again by performing a union operation between the column mesh and the room mesh (which now contains the cylinder block after the previous union operation).

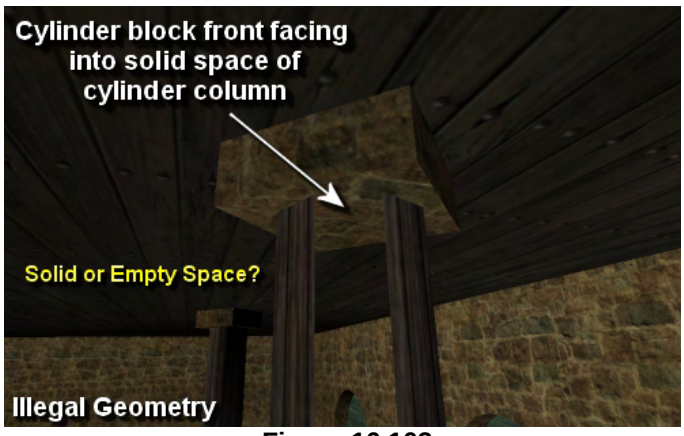


Figure 16.103

The results of this second union operation can be seen in Figure 16.104 where we note that the section of the bottom face of the pillar block that is facing into the cylinder's solid space is removed, and so too is the top face of the cylinder block. The cylinder block, the cylinder column, and the room geometry (walls, floors, ceiling, etc.) have now all been merged into one mesh with hidden surfaces removed.

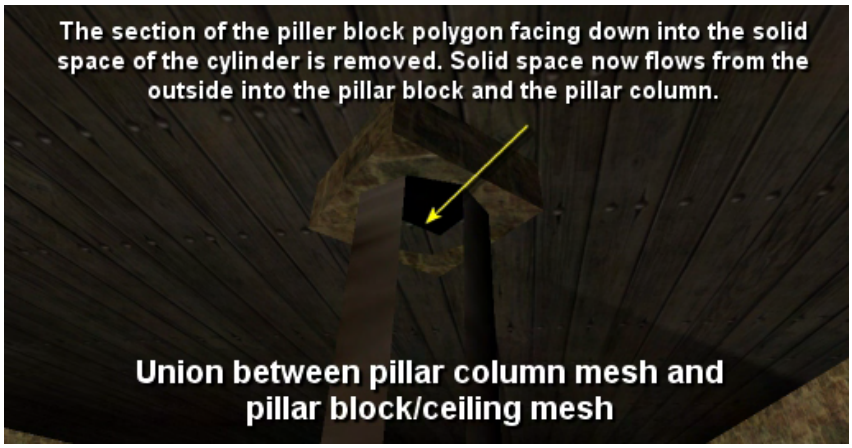
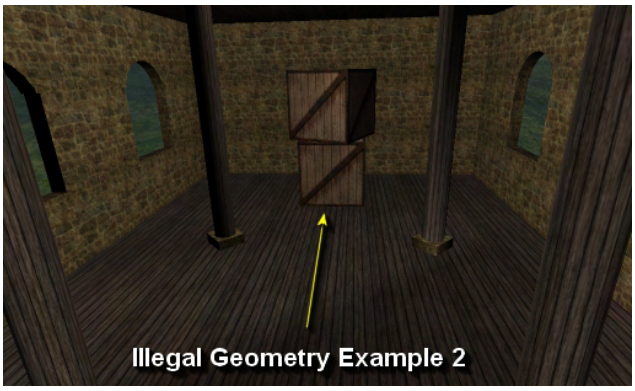


Figure 16.104

Once again we have made the front faces of the cylinder invisible so that we can see what has happened inside its solid space. Notice that the union operation has clipped a hole in the bottom face of the block which matches the dimensions of the cylinder column. Although it cannot be seen here, the top face of the cylinder has also been removed so that solid space flows from the outside, through the cylinder block and into the pillar columns. Solid space is still perfectly bounded by the backs of polygons and cannot leak out into empty space.

As Figure 16.105 shows, even if we select the all the cylinders and the room meshes and perform a union on them all to remove the hidden surfaces and correct these problems, illegal geometry can be caused by simply placing two crates in a room stacked on top of each other. This seems pretty harmless, but of course, if hidden surface removal (union) is not applied to these crates, the exact same geometry problems arise.



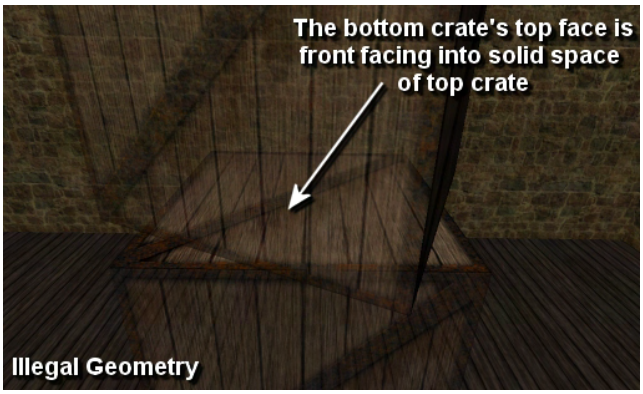


Figure 16.105

By making the polygons of the crates transparent in the rightmost image, the problem becomes immediately obvious. The top face of the bottom crate and the bottom face of the top crate are co-planar but facing in opposite directions. The top face of the bottom crate is facing into the solid space of the top crate and vice versa. Therefore, our HSR routines should also merge these two brushes/meshes together using a union operation to remove the offending hidden sections of the surfaces.

Figure 16.106 shows the result of performing a union on these two crate meshes into a single mesh.

We can see by making the faces of the top crate transparent that the entire section of the bottom crate's top face that was pointing into the solid space of the top crate has been clipped away and vice versa. These two crates are now a single mesh which is filled with solid space and whose outward facing polygons bound that solid space.

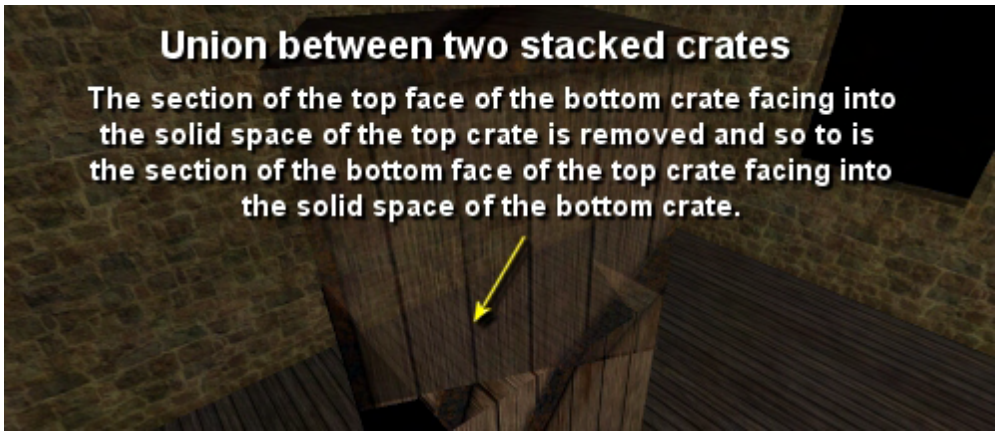


Figure 16.106

Of course, the same problem still exists between the bottom face of the bottom crate and the floor polygon of the room mesh on which it is resting. As Figure 16.107 illustrates, performing a union operation on our two crate mesh and the mesh of the room itself will remove the section of the floor that faces into the solid space of the crates and will remove the bottom face of the crate that faces into the solid space behind the floor.

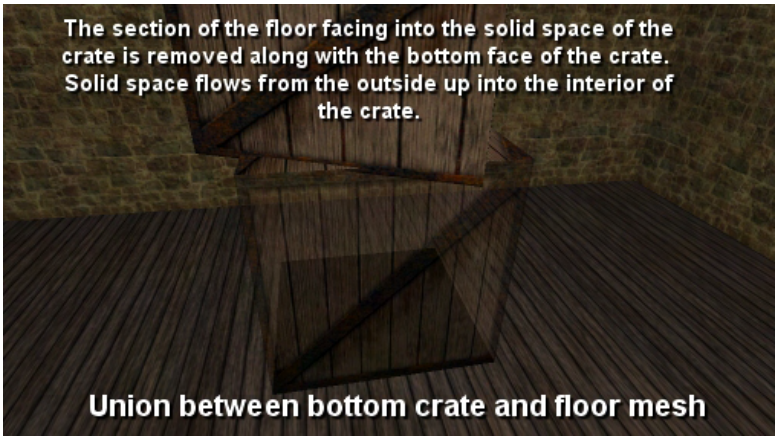


Figure 16.107

So we have seen how easy it is to create illegal geometry but we have also shown how in all these cases, HSR (union) comes to the rescue. The union operation is one of merging all the meshes that comprise the scene into a single mesh removing all hidden surfaces in the process. If the world editor/3D modeler of your choice does not support CSG and the union operation in particular, do not worry. We will discuss how to perform a union operation on your scene in the next section of this lesson and will also implement an HSR module in our BSP compiler in Lab Project 16.2. This HSR module will be invoked prior to the BSP compile to remove any hidden surfaces that exist in a multi-mesh environment.

Finally, before discussing CSG it is important to take a moment to strongly recommend that you spend a good deal of time thinking about which objects in your scene should be considered BSP geometry. Remembering that nearly every polygon you pass into the compiler will be used to create a node, we can imagine how placing a 10,000 polygon sphere in the middle of a room would create 10,000 nodes in the tree, each of which would split the polygons of the level into many tiny slivers. This would make for an extremely large tree that is slow to traverse and high on memory usage. Furthermore, it would raise the polygon count of the scene significantly.

It is quite common for only the core geometry of the level to be compiled into the BSP tree (e.g., walls, floors, ceilings, etc.) and for high polygon objects and room accessories such as furniture meshes and

décor to be assigned to the tree as dynamic/detail objects. For example, GILES™ has a property that allows you to flag a mesh as being a detail object which our BSP compiler will not compile into the tree -- it will simply write it back out to the final compiled file. When our BSP file is loaded, these detail objects can be loaded also and passed down the tree and assigned to the leaves in which they reside. This produces a much smaller and faster tree and is highly recommended in many situations.

> **Important Note:** To keep the tree small and stable, you should only be compiling your core geometry into the level. High polygon models used to populate the scene with furniture or décor should not be compiled and should be assigned to the tree as detail objects at run time.

Bearing this in mind, we might imagine how we could sidestep the entire illegal geometry problem in the previous examples by compiling only the original room meshes into a BSP tree. Once the BSP tree of the empty room has been compiled, the crate and pillar meshes could simply be assigned as detail objects to leaves in which their bounding volumes are found to reside. When these leaves are considered to be visible, the detail objects assigned to those leaves are also flagged as visible and rendered just as was the case with our previous tree types. If we were to take this approach, the only geometry from the previous example that would be compiled into the BSP tree is shown in Figure 16.108.

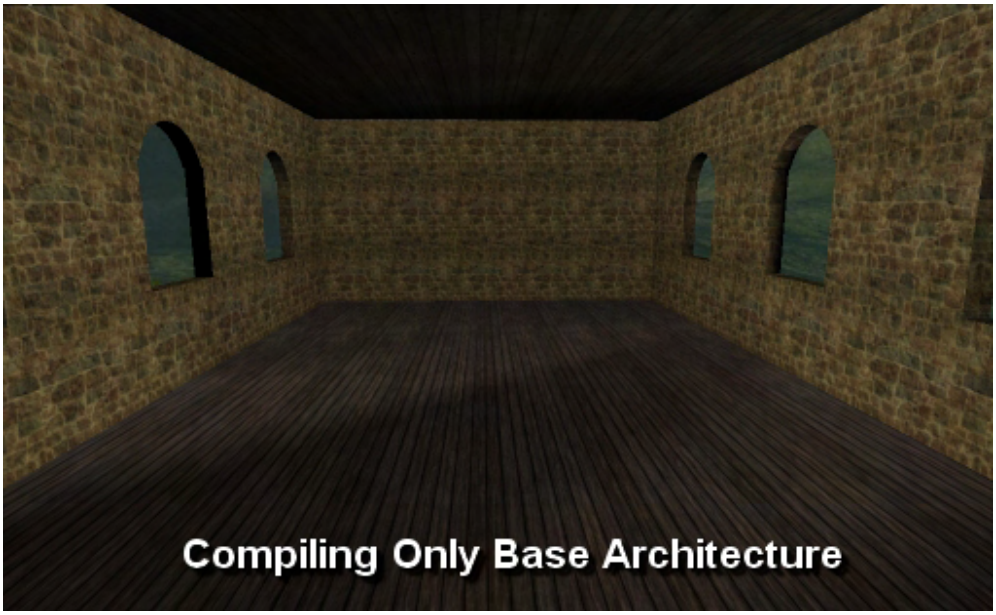


Figure 16.108

Of course, it is not always convenient or efficient to have to update the leaf assignments for objects that you know are static, so for simple static objects like crates and pillars, you may decide that you want to compile them into the tree, which means the HSR routines demonstrated above will have to be discussed and implemented. Fortunately, that is exactly what we are going to discuss in the next section.

16.5 Constructive Solid Geometry

In this section we will be examining the process known as Constructive Solid Geometry (CSG). Over the next several pages we will look at how to apply the BSP tree knowledge we have gained thus far when performing CSG and hidden surface removal (HSR). Our approach to CSG will require that the geometry we send through the compiler is *brush based*. This means that our level data must be made up of a set of enclosed hulls or primitives (e.g. cubes, cylinders, teapots, etc. – basically anything that is completely *solid*). This is the way that many of the most popular level editors work (e.g. WorldCraft™ or GILES™). As we alluded to earlier on in this chapter, hidden surface removal is used to ensure that our geometry is legal and is actually a logical extension of CSG. In fact, we basically get this support for free when we implement CSG functionality using our BSP Compiler.

There are many reasons why we would want to use CSG operations. One of the main advantages for anyone writing a BSP Compiler is the fact that we can perform Hidden Surface Removal (HSR) using the **union** operation. If you have ever tried to create a reasonably complex level using a package such as 3DStudio MAX™ and tried to compile it into a BSP tree, you will likely have learned that illegal geometry is quite commonplace. HSR takes care of most types of illegal geometry which crop up when designing a level. It simply clips away any parts of intersecting brushes which could cause the compiler to fail. The other advantage we achieve by adding support for CSG into our compiler or world editor is the ability to carve one brush from another using the **difference** operation. This allows us to easily carve doors or windows from a wall, or perhaps create other hollow objects (e.g. coffee cups, etc.) that retain their solid geometric properties.

We will discuss HSR in more detail a little later, but in order to do so we must first understand the basic principles involved in each of the 3 primary CSG operations.

16.5.1 CSG Operation Primer

Constructive Solid Geometry (often referred to as a Boolean Operation) is the process of building solid objects from other solids. The three CSG operators most often encountered are known as the Union, Intersection, and Difference operators. Each of these operators acts upon two objects and produce a single object result. By combining multiple levels of CSG operators, complex objects can be produced from any number of simple primitives. Due to the fact that we are basing our shape's 'solid' and 'empty' space on the information constructed by the BSP tree, we need not even ensure that these objects are convex to begin with.

The following set of diagrams illustrates the three CSG operations mentioned. As mentioned, although we are using simple primitives here the same principles apply with any shape we can create.

CSG Operations

We start off with our two brushes: a box and a cylinder. We can combine these two brushes in a number of ways in order to obtain various different results. You may have seen these operations elsewhere defined using their Boolean Operator names, so for future reference we have denoted these alternate names beside each operation heading.

Union (XOR)

This is the **union** of our two brushes. While it may seem as though they are simply placed on top of one another, in fact the union operation removes all polygons that fall inside the solid space of either brush. If we were to move the camera inside either of the brushes, we would see that any polygons (or fragments of polygons), which ended up in solid space are discarded. This operation represents the core of our HSR (Hidden Surface Removal) system. The **XOR** is basically the same as the Exclusive-OR operator seen in many programming languages. It simply keeps any area which does not share the same space (i.e. the parts of the box which lay outside the cylinder and the parts of the cylinder which lay outside the box).

Difference (NOT)

The **difference** operation is probably the most widely used of the three. Its purpose is to carve one brush from another. The difference (or carve) operation will result in different sets of polygons depending which brush you perform the operation with. For example, in this diagram we use the cylinder as the *cutting brush*, which basically subtracts the cylinder from the box. Only the parts of the box **NOT** inside the cylinder remain untouched. However if we turned the box into a cutting brush, it would leave behind the top half of the cylinder (since the bottom half lies inside the box, and is carved away).

Intersection (AND)

Here we see the **intersection** of our two brushes. The resulting geometry of the intersection operation describes only the area of the two brushes which were intersecting (i.e. only the parts of the box **AND** the cylinder which were intersecting remain untouched). This operation is not as widely used as the union or the difference operations, but is an alternative method of creating certain geometry if the other operations are not suitable. Some editors choose not to implement this operation at all.

16.5.2 CSG Principles

With the aid of the solid BSP leaf tree, CSG is not nearly as complicated as it may first appear. In this section we will discuss the theory behind the various CSG operators outlined above before we move on to examine how we might implement a complete CSG processor.

Figure 16.109

The first thing to realise when adding CSG to our applications, is that these methods will only work when the geometry is represented as *Brushes* (or *Solids*). What this basically means is that our level must initially be made up of various primitives such as cubes, cylinders, wedges, spikes, cones, spheres and so on. This is not as limiting as it first sounds. When we refer to brushes or solids we essentially mean anything which can be defined as having a solid space. That is, it is completely enclosed such that solid space cannot leak out into empty space. Thus any primitive that fits these criteria (such as the object shown in figure 16.109) can be a considered as valid input for each operation. The reason for this requirement will become clear when we begin to discuss how CSG actually works. It should be instantly apparent why the BSP tree is an ideal candidate for this job. Due to the fact that the very foundation of constructive solid geometry is based on the determination of solid space within any particular object, the solid / empty information provided by our BSP tree implementation will be of immense benefit to us.

As mentioned, the key to successfully implementing CSG, whether we have the aid of a BSP tree or not, is being able to determine which areas of our brush are solid and which are not. There are many problems associated with making this determination when we are not using a BSP tree, but fortunately for us, this will not be a concern.

In a short while, we will be describing how the various CSG operations actually work. But before we do this, it is vital that you understand exactly what we are describing when we talk about difference (NOT), union (XOR) and intersection (AND). So what we will do is have a brief refresher on bit manipulation which shows how the various operators (used in most programming languages) work on simple numbers. This should help us to better visualize the definitions we will be using throughout the remainder of this chapter.

Although we will be discussing the effect that these various operations have on bits and not on actual geometry at this point, it is helpful to imagine that each bit that is set to 1 is an area of space within the solid interior of the brush it is describing. The tables that follow will show how the areas of both of our brushes after the operation has been performed contributed to the construction of the final combined brush.

> **Note:** If you have any trouble understanding the bit operations, it may be helpful to start up the calculator that comes with Windows (or any scientific calculator which can handle binary numbers) and follow them through for yourself. Make sure that the calculator is running in scientific mode (View/Scientific for the Windows calculator).

The XOR (^) Operation (Union Equivalent)

Figure 16.110

The XOR operation can be a little tricky to understand at first. XOR compares each bit of the first set against the corresponding bit of the second set. Any bits that are equal in both sets (i.e. both equal to 0 or both equal to 1) will result in the corresponding bit in the output being set to 0. However, if either of the corresponding bits equals 1 while the other is 0, then the output bit will be set to 1.

The following bit tables should make this clear. We will be defining the bits as if they were areas of space occupied by a cylinder and a cube. What remains after the XOR operation will be the areas of space taken from the original brushes and used in the resulting brush (i.e. if a bit is set to 0 then the parts of either of the brushes which occupied this space will be discarded and not used in the resulting brush)

Cylinder Brush

1	**1**	**1**	**1**	**0**	**0**	**1**	**0**

XOR

Cube Brush

0	**0**	**0**	**1**	**1**	**1**	**1**	**1**

=

Resulting Brush

1	**1**	**1**	**0**	**1**	**1**	**0**	**1**

Keeping in mind that a 1 in the table above corresponds with an area of space occupied by the solid space in the brush it was describing, you can see that the areas of the cylinder which were sharing the same space as that of the cube will be removed in the resulting brush. It is important to take into account the fact that the result in these examples will describe how the two brushes are to be merged. So if the result contained a 0, then neither the section from the cube brush or the cylinder brush which occupied this particular space will be used in the final brush. This means that any part of the cube inside the cylinder and any parts of the cylinder inside the cube will not be used in the final brush (they are discarded).

Although it may not be clear at the moment, it may be worthwhile to know that this operator is the basis of the hidden surface removal techniques we will be discussing shortly.

The AND (&) Operation (Intersection Equivalent)

Figure 16.111

The AND operation is the simplest of the three. Put simply, if *both* of the corresponding bits in *both* of the sets equal 1, then the corresponding bit in the result will also equal 1. In any other case a 0 is the result. In figure 16.111 we see that only the parts of the two brushes which occupied the same space have remained.

As before, what remains after the AND op will be the parts of the original brushes used in the resulting brush.

Cylinder Brush

1	1	1	1	0	0	1	0

AND

Cube Brush

0	0	0	1	1	1	1	1

=

Resulting Brush

0	0	0	1	0	0	1	0

We can see that in the AND operation, the majority of the space described by each source brush has been discarded. Only the areas which overlapped and shared the same space now remain.

The NOT (&~) Operation (Difference Equivalent)

Figure 16.112

The NOT operator is not quite the same as the AND, OR, or XOR operators. When we use a logical NOT in a programming language it usually returns 0 if the value passed in is non-zero, and 1 if the value passed in is zero. The same logic applies to a bit-wise NOT. All of the bit values are switched from 1 to 0 and vice versa. Recall from our earlier definition of the difference operator that the resulting output of this operation depends on which brush was used as the carving brush. In this case, the space and polygons described by this brush need to be inverted such that we retain the portions of the carving brush that falls into the solid space of the opposing brush, and likewise retain the parts of the second brush that fall into the *empty* space of the carving brush. Because of this need to reverse the situation found in the intersection operation, this procedure basically performs a NOT operation on the carving brush first (but not on the brush we are carving from) and then an AND op on these two sets of bits. This means that we are essentially describing a NOT-AND operation (or NAND).

Cylinder Brush (Carving Brush)

1	1	1	1	0	0	1	0

NOT

Cylinder Brush after NOT

0	0	0	0	1	1	0	1

AND

Cube Brush (Carved Brush)

0	0	0	1	1	1	1	1

=

Resulting Brush

0	0	0	0	1	1	0	1

Here we see that the resulting brush contains only those parts of the original cube which were not occupied by the cylinder. If we wanted to carve the cube from the cylinder instead, we would apply the NOT to the cube rather than the cylinder.

Hopefully you now better understand what each of the CSG operators do. One thing that you will find about constructive solid geometry as we move forward is that that understanding what each operation does is actually harder than implementing the operators themselves. Since we will of course be working with real geometry, the 'area of space = a bit' analogy we used in the previous examples does not ultimately describe the techniques we are going to implement even though they clearly demonstrate the results we should expect.

In the next section we will begin to look specifically at how these CSG operations will function in real world situations, using physical geometry as input.

> **Note:** If you would like to see a practical demonstration of each of these operations in a real world application it is recommended that you obtain and install a copy of the GILES™ world editor from your class CD or download area. In this application you will be able to try out each of the CSG operators described above, using any manner of varying brush shapes that can be created. If you need help with using the GILES™ application, a full manual is included in the installation package, available from both the start and help menus.

16.5.3 Performing CSG with Geometry

At this point we are already well versed in BSP theory and we should know exactly how they subdivide the space within an object into solid and empty areas. Although when working with CSG we will usually be compiling only small subsets of our level at any given point, remember that the compilation principles used are exactly the same as those employed when compiling a multi-thousand polygon scene.

Before we begin to process any geometry for the purposes of our chosen operation, we first need to be able to determine which areas of space within each of the brushes in our scene (cube, cylinder, etc.) are solid and which are empty. In order to achieve this, we will first need to compile a unique BSP tree for each brush that we would like to take into consideration during the operation. Although it is not strictly necessary for us to compile each of these trees in advance – and in fact it is probably a good idea to only compile each tree as and when we need it – we must ensure that this information is available to us at some point during the procedure.

With this solid and empty spatial information to hand, we are able to implement each of our CSG operators using a simple series of clipping and removal operations.

16.5.4 Implementing the CSG Operators

As the basic principles behind the implementation of each of the CSG operators are the same, we will begin by examining the **union** (**XOR**) operation which we will use as the foundation for understanding how the other two remaining operators function.

The Union Operation

As we know, the union operator is intended for to be utilized in cases where we would like to merge two brushes together. In order to achieve this we must first clip each of the brushes' polygons to the others compiled BSP tree, and remove any fragments of those polygons that may fall into the solid areas of either brush's tree.

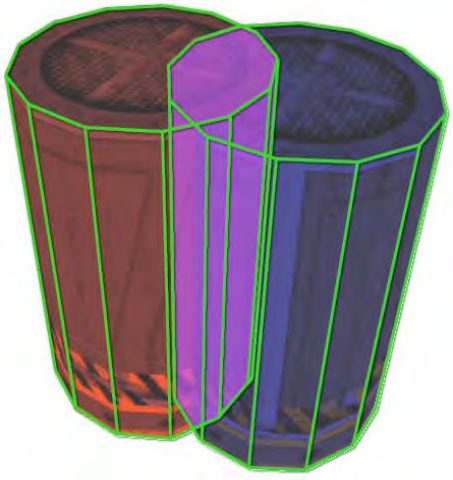


Figure 16.113

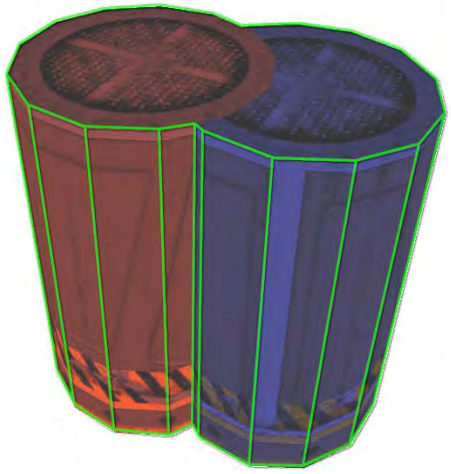


Figure 16.114

Figure 16.113 shows two intersecting cylinders. We can clearly see that we have begun this process using two separate convex hulls (the cylinders themselves) both of which have their own solid space (the red and blue areas). We can also see that where the cylinders overlap, we have an area of space that is shared by both hulls (the purple area). Figure 16.114 shows the outcome of the union operation, where the sections of each hull that lay inside the others' solid space has been removed. The result is a single solid shape which has had any and all interior polygons split and removed. One thing that you may have noticed about the resulting brush in the above figure is that the top faces of the cylinder seem to have remained untouched. Obviously if we were to remove both pieces of the top polygon that fall inside each other's solid space we would be left with a hole in our new solid. The specifics about how to handle this case are covered shortly, but for now just concentrate on how the boundaries of the objects (shown as green wire frame) have been merged together to form one continuous mesh.

Given the basic concept of this operation, let us take this background knowledge and apply it to two simple cubes using the BSP Tree. The first step is of course to build up a BSP Tree for each of the cubes (we will ignore the top and bottom polygons of this cube in order to keep the examples clear, but the same principles apply). Figure 16.115 depicts a top down view of a cube brush and figure 16.116 shows the resulting compiled solid BSP leaf tree.

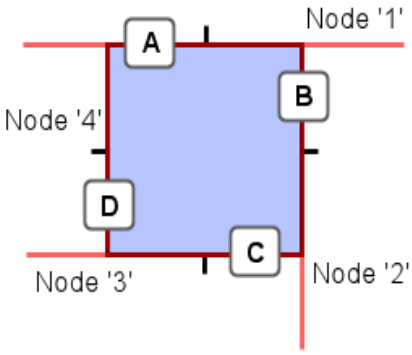


Figure 16.115

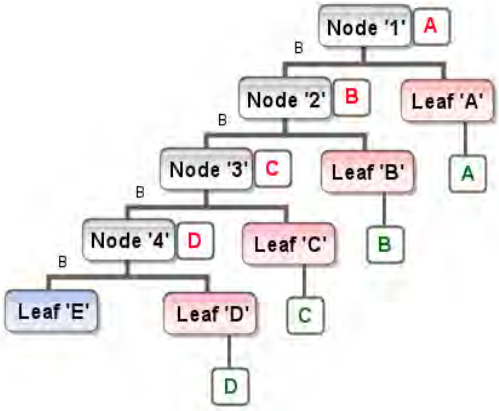


Figure 16.116

The process used to construct this tree should already be familiar at this point so we will not go into great detail about how we ended up with the hierarchy shown. This tree is quite unbalanced and certainly very simple, but it will benefit us to start small. As in each of the prior examples, the letters in figure 16.115 denote the original polygons that bound the solid space inside the cube. As illustrated in figure 16.116, we can see that we end up with just one solid leaf at the bottom of the tree with each of the remaining empty leaves attached to the front of each node. With this information available, we can start to actually perform the union operation. We will use a second identical cube to perform the op with, since the BSP tree constructed for each cube will be the same.

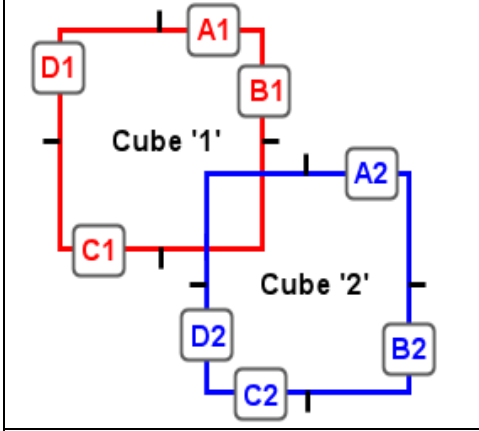


Figure 16.117

Figure 16.117 shows our two identical cubes and how they intersect. We will need to perform the same operation on both cubes (i.e. removing the areas of the first from the solid area of the second and vice versa), but we will begin by clipping cube 2's polygons against those of cube 1. To perform the union operation, we must pass every polygon contained within the first cube through the tree of the second cube, splitting the polygons against node planes as we recurse. Whenever a polygon fragment ends up in a leaf which describes solid space, it will be discarded. Once these steps have been completed for the first cube, we reverse the process and perform exactly the same operation with the second cubes polygons. Let us walk through the example step by step. We will refer to the cube 1's polygons as A1, B1, C1 and D1, and likewise the cube 2's polygons as A2, B2, C2 and D2. In the first iteration of this process we do not actually use the polygons of cube 1 at any point. This is the key to this entire process. At this point we are simply classifying and splitting the polygons of cube 2, against the nodes of the first cube's tree as we traverse.

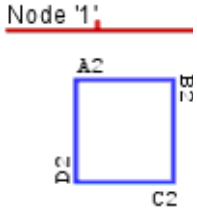


Figure 16.118

First we classify all of the polygons of cube 2 against the root node (node '1') of the tree we are clipping against. As we loop through each polygon, we can see that as we test them individually, each one is found to be completely behind this node. Just as with the actual BSP compilation process, each of these polygons should therefore be added to a *virtual* back list for this node (i.e. it is not physically attached to the node – we keep two separate lists called 'Front' and 'Back' and store each polygon in this list depending on where it lies in relation to the nodes plane. We will see why we do this in a moment). Once all of the polygons of the cube have been classified against this plane and added to the relevant list, we then step to the next node in the tree, passing the appropriate list of polygons. Since there are no polygons in the front list at this time, there is no need to step into the node/leaf indicated by node 1's front pointer. We do however have polygons stored in the back list so we pass this list down the tree to node 1's back node/leaf. (We will explain what happens if this is a leaf shortly).

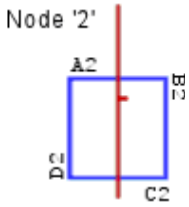


Figure 16.119

We have now passed all of our polygons to the node described by Node 1's back pointer, which happens to be node 2 in this case. As before, we start classifying polygons against the current node's plane. Starting with polygon A2, we find that this polygon is spanning the node plane and should be split. This gives us two separate polygon fragments -- one behind the plane (A2a) and one in front of the plane (A2b). We then add each respective split fragment to the relevant front or back list and discard the original polygon. Now we move on to polygon B2 which is completely in front of this node. Thus, it is added to the front list. Polygon C2, like polygon A2, is also spanning the plane and again must be split with the resulting fragments being added to the relevant list. Again, the original polygon is deleted. Finally we classify poly D2 and find that it is totally behind the plane. It is also therefore added to the back list.

At this node, we actually have polygons in both our front and back lists. We will deal with the front list first. Our front list contains part of the original A2 polygon (labelled A2b), part of the original C2 polygon (named C2b), and the complete polygon B2. As before, we need to pass this front list into

whatever node 2's front pointer contains, which in this case is an empty leaf. Because the union operation only discards polygons which end up in solid space, all of the polygons in this front list have survived the operation and as such, can be added to the final resulting brush at this point.

Our back list contains the fragments A2a, C2a and the in-tact polygon D2. This list is then passed down the back of node 2 and into node 3. We begin the process yet again. In figure 16.120, our split polygon fragments are shown in green.

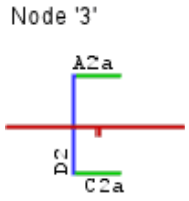


Figure 16.120

> **Note:** Remember that we are only testing against the polygons which actually survived to this point and were passed down to this node.

We start at the beginning of our list and find that the fragment A2a is completely behind the plane and is therefore added to the back list. Then C2a is added to the front list. D2 is spanning node 3's plane so it is split into two fragments (D2a and D2b) where D2a is added to the front list and D2b to the back.

As before, we have some polygons in our front list, so we pass this to whatever is in the front pointer of node 3. This happens to be an empty leaf, so whatever is in the front list has survived and can be added to the final resulting brush as before. Our back list also contains some surviving polygons, so it will be passed down the back of node 3 which directs us to node 4.

In node 4 we now have only two poly fragments that remain (A2a and D2b). We classify each of these and find they are both behind the plane and added to the back list. As in the example of node 1, we find there are no polygons in our front list, so there is no need to go recurse into the front. We do however have these two remaining polygon fragments in our back list, so we pass it down the back of node 4. However, remembering that we can distinguish solid leaves as being any attached to the back of a node this time around we find that we fall into a *solid* leaf. Since the union operation discards any polygons which make it into solid space, these polygons will *not* be added to the final brush.

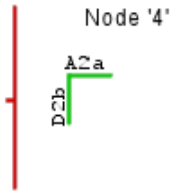


Figure 16.121

Figure 16.122 depicts the polygons that actually made it into the final brush. You will notice that the original polygon C2 has been split, and was added as two separate fragments (even though neither was deleted). Unfortunately there is no way to avoid this during the traversal process, but there are many techniques we can employ to reduce the number of splits in our final brush after processing has been completed. These will be discussed as we cover the implementation of our final CSG processor.

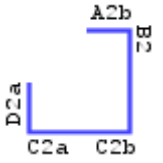


Figure 16.122

As mentioned, we have to perform this process for both brushes. So the next time around we will have to pass all of the ***second*** cube's polygons through the ***first*** cube's mini-BSP Tree in the same way as we have done with the first. But how can we do this now that we have clipped and removed most of the polygons? Actually this will not be a problem. Remember that we have already compiled the Mini-BSP trees for each brush, before the CSG process begins. This means that even though the polygons of cube 1's tree have been altered, the tree of this cube is still completely intact in its previous state. For the purposes of CSG, the polygon information that is stored in the tree is essentially irrelevant, all we need to pay attention to are the node planes, and the leaves themselves. While we will not step through the

process again for the second cube (as you should now be able to do it for yourself) the following figure illustrates the process.

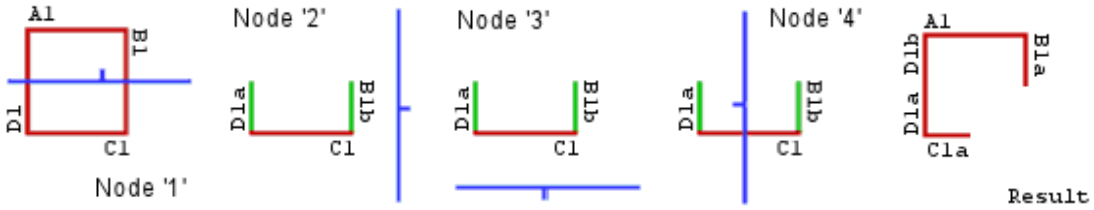


Figure 16.123

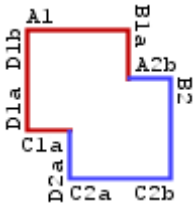


Figure 16.124

We see now that the final results of both clip operations yielded the hull shown above. We can also see exactly where each polygon came from. Although this jumble of nodes, planes and polygons may be a little confusing at first, by referring back to Figure 16.124 and following the process through step-by-step for each brush, you should start to understand exactly what is going on at each stage through the tree traversal.

We are fortunate in that we are able to reuse much of the functionality of our BSP tree code when performing CSG – for instance, our polygon classification and splitting routines. This means that with a little bit of planning, we can actually add our complete BSP tree clipping routine in just one function thanks to recursion. While some people operate under the impression that recursion is a large performance penalty, the reality is that most C++ compilers can generate much faster code than the average programmer trying to implement a manual stack. We do not really have to face the typical problems like stack overflows, because the depth of the Mini-BSP Trees will be relatively small.

One thing that we did not encounter in the previous scenario is what happens when we find a coplanar test case. As it turns out, the on-plane case does present us with a small problem when performing a CSG operation. Let us take a look at this problem now.

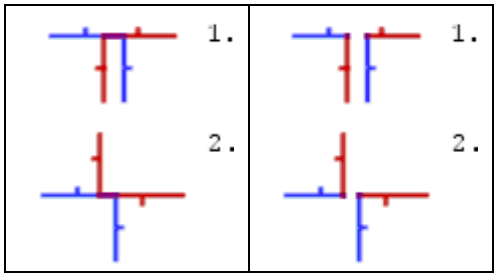


Figure 16.125 **Figure 16.126**

Figures 16.125 and 16.126 demonstrate the ‘before’ and ‘after’ of two separate cases which include polygons that are coplanar with the nodes of the tree we are testing against. In the top case (case 1), there are two polygons which share the same plane and in this case both face in the same direction. In the bottom case (case 2), the polygons are on the same plane but are this time facing in opposite directions. Let us examine case 1 first. By referring to figure 16.126, we can see the outcome if we were to automatically pass these on plane cases down the back of the node with which it is sharing a plane.

The reason this would happen is that, as we know, eventually these polygons would make their way into a solid leaf and would never be added to the final resulting brush. If you look at the colours of the polygons in case 1 in Figure 16.126, you will realise that once the CSG processor has run its course, the two lower polygons will eventually have been removed because they reside in solid space, but the gap still remains. So we end up with a hole in our hull where polygons were previously overlapping. This is clearly not a good situation, and will more than likely result in our final BSP compile failing due to illegal geometry (i.e. space leakage).

However, in case 2, we find that passing the coplanar fragments down the back of the tree is correct. If you refer back to the union operation, you will realise that the two areas of solid space have been merged and that this is still a totally valid hull (all hidden surfaces have been removed). This is a crucial part of the HSR algorithm which will take place before final BSP compilation, so we are actually fortunate in this way to be able to simply test the normal of the poly against the plane of the node. If it is facing in the opposite direction to the node, then we can simply add this poly straight into our back list as we would do for any other poly which is behind the plane. This is virtually identical to the method we used to handle the coplanar cases during the construction of both the node and leaf BSP tree earlier in this chapter. Recall that we test to see if the polygon normal is facing in the opposite direction to the node or not. This also works for the other operators (such as difference and intersection) but we will discuss those a little later in the lesson.

So the only case we have to deal with when handling the coplanar cases is that in which both the polygon normal and the node's plane are facing in the same direction.

The solution to this problem of the hole in the resulting hull is relatively simple. We already know that we perform the clipping first on one brush, and then on the other. All we have to do is to allow the clipping routine to pass these cases (where both are facing in the same direction) down the back list for one brush, but when we reverse the operation and clip the other brush, make sure that it is passed down the front list instead. Again, refer back to case 1 in Figure 16.126 where we can see the result if we were to send both fragments, from both brushes, down the back of the clipping node. In that case, both fragments are removed. But if only one of the fragments is removed, by passing only one of them down the back and the other down the front, we can see that the one sent down the front will be added to the resulting brush and the gap will be closed. Earlier in this section we demonstrated the result of the union of two cylinders in figure 16.114, in which only one of the top faces was clipped, and the other remained in-tact to fill the hole. This is an ideal demonstration of this process in action.

Fortunately, as mentioned, this same process applies to all of the CSG operators that we cover in this chapter. This is nice because it means that for these other operators we can follow exactly the same steps we have outlined here. For the most part, we need only to decide whether we want to discard the polygons which end up in solid space or those that end up in empty space.

For more information on how best to handle the on-plane case during implementation, it would be a good idea to refer to the workbook for this lesson to see the approach that we have used.

The Intersection Operation

Due to the fact that the implementation process is the same between each type of operator, we can simply now discuss how we might adapt the techniques we learned in the union operator for use with the **intersection (AND)** operation.

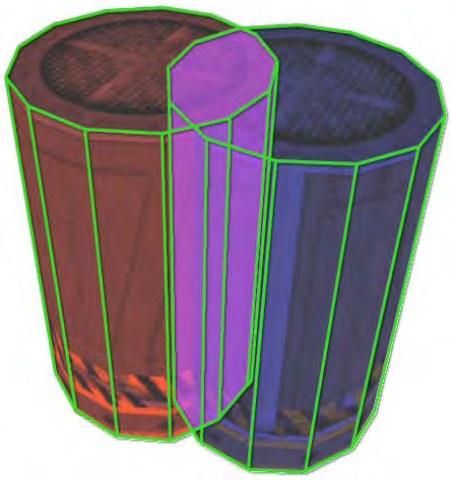


Figure 16.127

Figure 16.128

The intersection operator works in much the same way as the union operation did. In Figure 16.128 we can see that in this situation, everything that ended up in solid space has survived as opposed to being removed. There is not any significant difference in the way this works: all the on-plane cases remain the same, and everything is still split and sent down the relevant side of the clipping node as before. The only difference is that in order to produce the result we are looking for we need to discard any polygons which end up in empty space rather than in solid space as we did previously. Thus only polygons that end up in solid space are added to the resulting brush. Just as before, we perform the operation with one brush's polygons against the other brush's BSP Tree, and then reverse the process using the other brush's polygons against the first brush's BSP Tree.

For completeness the following diagram shows the result of this operation using our cube example from the last section. The step-by-step diagrams would be identical to those we saw in the union case and have therefore not been included here.

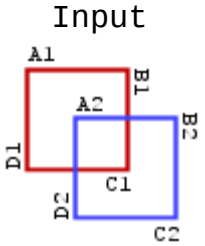


Figure 16.129

Figure 16.130

The Difference Operation

The final operator we will discuss is the **difference (NOT)** operator. It works in a slightly different manner than the other two operators, due to the fact that it is dependant on which brush we want to use to perform the operation with.

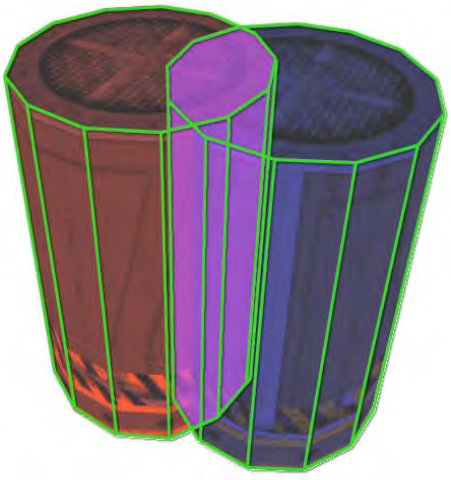


Figure 16.131

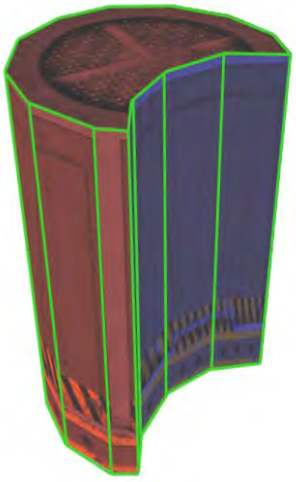


Figure 16.132

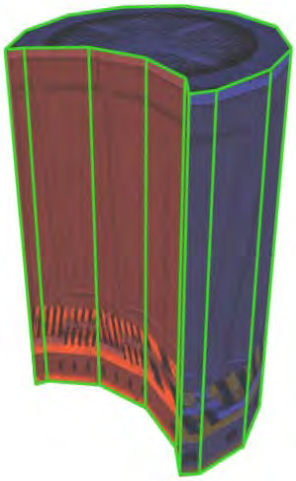


Figure 16.133

There are actually a number of aliases by which this operation is known, and you may encounter them in other contexts. Very often you will hear the difference operation referred to as a NOT, NAND, carve or subtract operation. We can see that Figure 16.132 shows the result after the blue hull from figure 16.131 has been subtracted or carved from the red hull. By the same token, Figure 16.133 shows the result after the red hull from figure 16.131 has been subtracted or carved from the blue hull.

It is clear then that the outcome of this operation does indeed depend on which brush we carve from the other. If you look back to our earlier definition of ***NOT***, we can also see that in our literal definition of the NOT / NAND operation, the outcome depended on which set of bits we swapped using the bit-wise NOT operation.

The interesting part is how we 'NOT' the brush we want to carve with, in the same way we did with the bit manipulation we performed earlier. Just as the NOT inverted the bits, we will invert all of the polygons and their normals before compiling the mini-BSP tree for this brush. In doing so, solid space becomes empty space and empty space becomes solid space. We will then perform the AND operation as shown in the intersection operation previously discussed (i.e. keep anything which ends up in solid space in either of the brushes).

Although this operation is very similar to the intersection op shown previously, we will step through the entire process as we did with the union since it is very important that you fully understand the process. Keep in mind that the BSP tree has been built in a different fashion this time around (i.e. it is inverted). To help illustrate this, the new tree which was built from the *inverted* brush polygons is shown next.

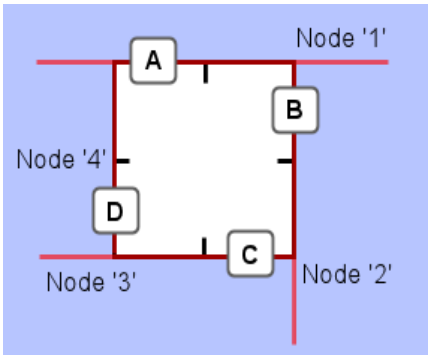


Figure 16.134

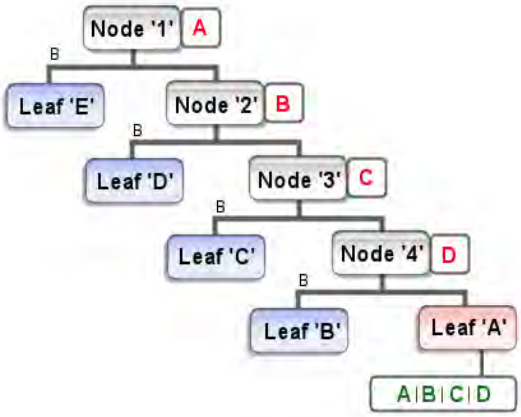


Figure 16.135

Figure 16.135 shows our inverted tree structure in which you can see that each of the brush polygons have been. Just remember that we cannot generally get away with inverting only the normal; we also have to invert the winding order of the polygons themselves if we want these polygons to render correctly when attached to the final resulting brush. We will explain how to reverse the winding order in the accompanying workbook for this chapter.

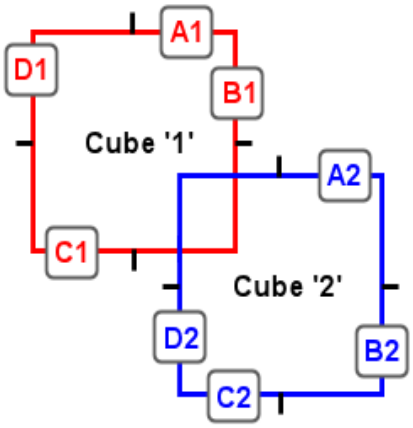


Figure 16.136

As before, we are going to use the same two cube example. In this example scenario we are going to be carving cube 1, from cube 2. Remember that cube 1 has been inverted and its mini-BSP tree has been built from the polygons in their inverted state. This means that the node planes will point in the opposite directions as shown in the figures above. Also remember that we have done nothing to the second cube just yet. We do not need to invert the polygons of any brush we are carving *from*, only the brush we are carving *with*. You can refer back to figures 16.135 and 16.136 for the definition of the second cube's tree if needed. To perform the difference operation properly, we need to do exactly the same thing as we would when performing an intersection (AND) operation assuming we have built this inverted tree. This means we will delete everything that ends up in *empty* space. This will be achieved in the same way in which we performed the clipping in our step by step union example with the exception of the differing polygon discard rules. For future reference, it does not particularly matter which brush we start with, but this example will focus only on clipping the second cube, to the tree of the inverted cube '1'. Let us now examine the rest of the process.

Looping through in alphabetical order, we find that all of our polygons which are being tested against node ‘1’ are in front of the plane, and as a result they are all added to the front list. There is nothing currently in the back list so we do not need to pass anything down the back of this node. Thus, we continue down the front, passing our newly built front list.

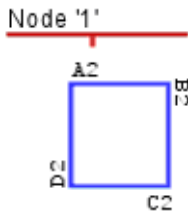


Figure 16.137

The front leads to node 2 (see figure 16.135) and we again classify our polygons against the node. We find that polygons A2 and C2 need to be split and their fragments are added to the relevant front/back lists. (Remember that we delete the original poly at this point). B2 is placed in the back list and D2 also goes in the front. Although our function would usually traverse down the front of the node before dealing with the back list, we can do it now to save confusion due to the recursive nature. The back list (containing A2b, B2 and C2b) is passed down the back of the current node and we find ourselves in a solid leaf. Since the difference operation keeps any polygons or fragments of polygons which end up in solid space, these polygons can be added to our final brush. The front list is passed down the front of the node which leads us to node 3.

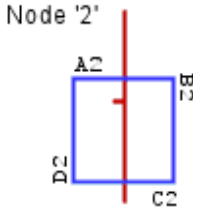


Figure 16.138

The diagram above shows the polygons that were passed to node 3. Again we classify all of the polygons and find that C2a is added to the back list and D2 is split into two fragments on either side of the plane. D2a is therefore added to the back list and D2b is added to the front list along with A2a which is also in front of the plane. As before, the back list is sent down the back, and again we find a solid leaf and add these surviving polygons from that back list to our final brush. The front list goes down the front of the node, which takes us finally to node 4.

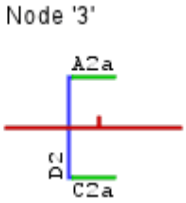


Figure 16.139

Classifying the remaining polygons, we find that no polygons are added to our back list and that both are added to the front. Our recursive function then attempts to pass this list down the front of the node, but we find that there is an empty leaf here. Since this operation discards any polygons which end up in **empty** space, these final two poly fragments are deleted.

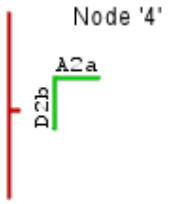


Figure 16.140

The final resulting brush is shown in the figure 16.141. We can see that, as we are clipping the second cube, this figure depicts exactly the same resulting fragments as those from cube 2 that survived during our union operation. This clearly demonstrates that swapping the rule about whether we delete what is in empty space or in solid space really does work as expected.

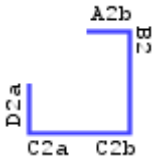


Figure 16.141

While this part of the difference operation was fairly obvious, the next bit is where the main differences lies. We can see the steps involved in clipping the first cube (the inverted brush) against cube 2 in figure 16.142. You may notice that the diagrams are identical to those we saw with the union operation. You should be able to work your way through the process with the aid of the following illustrations without much trouble. Be sure that you understand exactly why we get the results that we do.

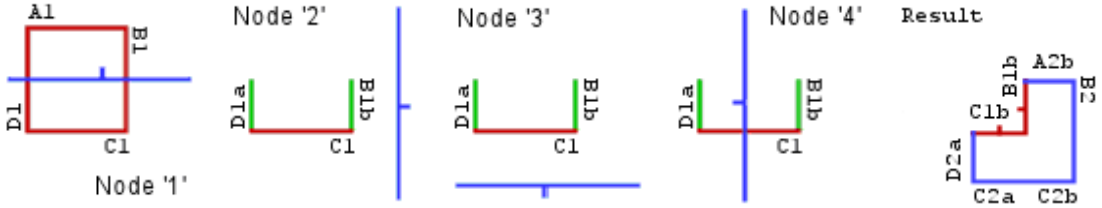


Figure 16.142

If you carried out this process on paper, you will hopefully have arrived at the result shown above. The red polygons will be all that remain of cube 1. Due to the fact that we inverted cube 1's polygons before we built the mini-BSP tree for that brush, you can see that they are now automatically pointing in the correct direction. They also close up the gap which would have been left cube 1's exterior hull.

> **Note:** Bugs in CSG processors are very difficult to track down. If there is a problem during implementation, it is very rare that the problems you see in the resulting geometry will have any bearing on the problem inherent in the code. For this reason it is important that you understand the operations completely.

Additional Notes about the Difference Operator

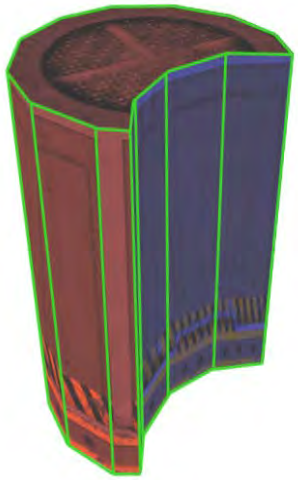


Figure 16.143

In an ideal world, the difference operator would fit neatly into the concepts we discussed in our 'area of space = a bit' examples. However, in some cases we may not be able to invert the brush data before compiling the mini-BSP tree. For future reference, we will look at one possible solution for this case. When we are performing the difference operation, we obviously still have to clip one brush to the other and vice versa in the same way we always have. For this explanation we will use Figure 16.143 as the target result we would like to obtain.

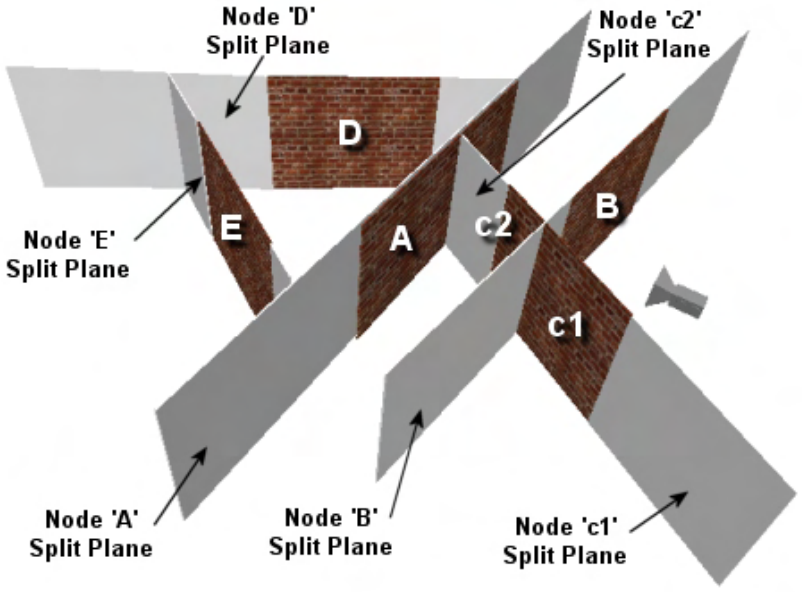
In this case we will assume that we are not able to invert the brush data before rendering, or for whatever reason, cannot invert the BSP Tree data after it is built. The solution we can employ is that when we clip the red hull's polygons using the blue hull's BSP Tree, we **delete** any polygons which end up in **solid space**. However, when we then clip the blue hull's polygons against the red hull's BSP Tree, we **retain** any polygons which end up in the red hull's **solid space**, and **delete** anything which ended up in **empty space**. There is one final step: the polygons that remained of the blue hull (which we can see in Figure 16.143) must be inverted before being added to the final brush so that they point outwards and seal up the hull. By switching the rule between the two traversal / clipping procedures in this way, we are essentially getting

the same result as inverting the tree itself. In theory however, this is not as clean as simply performing a NOT on the entire carving brush. Although either solution is a valid one, it is really more a matter of preference than necessity in many cases.

16.5.5 Conclusion

We covered a good amount of new information in this section that should be fully understood before continuing on. It would be a good idea to try some examples out on your own: perhaps an 8-sided cylinder against a cube. You can also experiment with these operations in the GILES™ world editing program. Try some of the operations on cubes, cylinders, and spheres and see what sorts of results are generated by the CSG processor under different circumstances. It will help to have a visual understanding of what it is we are aiming for as GILES™ itself uses all of the operations we have discussed here in this chapter.

Workbook Sixteen:
Binary Space Partitioning



Lab Project 16.1: Using BSP Node Trees for Alpha Rendering

The focus of this lab project will be on writing a class that will compile a BSP node tree from a list of input polygons. We will also expose methods that allow the application to render its polygon data in a back to front order with measures taken to increase efficiency where possible.

Unfortunately, although there is some degree of shared functionality, our BSP node tree class will not be derived from ISpatial tree or CBaseTree. This is mostly due to the fact that many of the methods exposed by ISpatialTree make no sense in the context of the BSP node tree. Most obviously, the node tree has no concept of leaves like the other tree types, and thus many of the ISpatialTree methods which work with leaves (e.g., ISpatialTree::CollectLeavesAABB), have no analogous concept in the BSP node tree. In the end, many of the method of ISpatialTree would have to be no-ops in the derived class if we wanted this to work. Our decision was that this was really not a good way to go. When an application sees a method exposed by an interface, it generally expects that method to do what its name describes. In the case of the BSP node tree, there are just certain methods which would make no sense due to the way it is constructed.

The rendering system in CBaseTree is also inappropriate for our BSP node tree. The CBaseTree rendering system worked by recording runs of adjacent triangles with like attributes so that we could render as many triangles as possible in the same subset with a minimum number of draw calls. We know that batch rendering in this way is always desirable for good performance. However, the last thing we want our BSP node tree to do is choose its polygon render order based on subsets. The whole point of us implementing the node tree in this lab project is so that polygons are rendered in a very specific back to front order based on view distance from the camera. Therefore, even if rendering in a back to front order means we have to change textures and materials on the device prior to rendering each polygon, there really is nothing we can do to avoid this. The polygons have to be rendered in that order for the blending to occur correctly and the whole purpose of implementing the BSP tree in this lab project is to ascertain that very order at run time.

Although the BSP node tree is a spatial tree it is being used in this lab project for a very specific purpose; it is not being used as a generic spatial manager. In fact, the BSP node tree would make a very poor choice of a spatial manager for your entire scene due to the fact that a polygon would be stored at every node in the tree, producing a large and deep tree. This is another reason we have not derived the BSP node tree from ISpatialTree. It is not to be considered as a viable alternative to our quad-tree, kD-tree and oct-tree classes for general scene management. Remember, the BSP node tree is being used to perform back to front rendering as its sole priority. As we currently have no need for this feature when rendering our opaque polygons, using the BSP node tree are our main spatial manager would result in unnecessary performance loss.

Thus, the CBSPNodeTree class is not a replacement class for our previously implemented spatial managers; it is to be used alongside them to handle only the rendering of the alpha polygon data. Our CScene object will now contain an ISpatialTree pointer and a CBSPNodeTree pointer as member variables. All polygon data will be rendered with the spatial tree, just as before. Transparent polygons will still be registered with the spatial tree as well, but their visibility status will be set to invisible. This means the spatial tree will compile the transparent polygon data into the tree, but not compile it into the vertex and index buffers used to render the tree. The transparent polygons will not be rendered by our spatial tree, but they will still be collision tested. A separate copy of the alpha polygon will then be added to the BSP tree. The BSP tree will contain all the alpha polygons after the load process is completed. Our

CScene::Render function will first ask the spatial tree to render itself (so all opaque polygons get rendered into the frame buffer with depth writing enabled) and then ask the BSP tree to render itself. The BSP tree will traverse its list of polygons in a back to front order and render its polygons with depth writing disabled and alpha blending enabled.

While it may seem that we are essentially throwing out batch rendering when it comes to rendering our BSP node tree, in practice things are not quite as bad as they first seem. For example, one might devise a render strategy for the tree that simply steps through each node in back to front order and calls IDirect3DDevice9::DrawPrimitiveUP for each polygon stored there. This would involve calling DrawPrimitiveUP for every polygon in the tree, and we already know how inefficient that would be from our discussions in previous lesson. Of course, one might assume at first that there is no better way to do it because we may end up having to switch attributes between each polygon, thus rendering a single polygon at a time anyway However, it is not uncommon for a game level to use only a handful of effects/textures for all its transparent surfaces and as such, there is a good chance that all the alpha polygons in our tree may belong to the same subset, or at the very least, collectively belong to only a small number of subsets.

This means that there is a very good chance that, as we are walking the tree in a back to front order and stepping from more distant nodes into nodes that are nearer the camera, the polygons in those nodes will share the same attribute. If we can collect the triangles in a back to front order into some temporary buffer during the traversal, the further triangles will be at the beginning of the buffer and thus will be rendered first when we send the buffer of triangles into the DrawIndexedPrimitive method. Of course, as soon as we collect a polygon during our traversal that does not share the same subset as the polygons we have previously collected in the buffer, we will have to render the buffer and then start collecting for this new subset. That is, every time we encounter the need for a subset change when collecting the triangles, the triangle buffer will have to be flushed. This system will allow us to (hopefully) get the benefit of some amount of batch rendering when adjacent nodes contain polygons with matching attributes. Ultimately, this allows us to reduce the number of draw calls.

Before we start to examine the source code to Lab Project 16.1, we will first examine the two basic components to the system. These components are the building component that compiles the tree and the render component that formats the compiled tree’s polygons data into a renderable format. The latter part will have methods to traverse the tree and render that data as efficiently as possible. Of course, these components are not new to us due to the fact that all the spatial trees we developed in the previous lesson had these same two components.

The Compilation Process

Although the CBSPNodeTree class is not derived from ISpatialTree and CBaseTree, we have tried to keep its operational flow and the names of its methods the same as their ISpatialTree counterparts. That is, the construction of the tree will be done in basically the same way.

The application will register polygons with the tree by calling the CBSPNodeTree::AddPolygon method. Our BSP node tree implementation does not need to support detail areas and dynamic objects like our previous spatial managers since there are no leaves to store this data. The focus of this class is thus very simple: compile a tree of static polygon data that can be traversed in a back to front order at run time.

After the application has registered any static transparent polygons with the CBSPNodeTree the CBSPNodeTree::Build function is called (another process that is familiar to us). This function starts the recursive process of building the BSP tree and storing the CPolygon structures in their nodes. As discussed in the accompanying textbook, each node will contain all the polygons that entered that node during the build process that were found to lay both on the node plane and that have the same frontspace.

Our previous spatial trees had the option of being constructed using polygon splitting, with the no-clipping case often being the favored approach to reduce the number of new polygons introduced by the compilation process. However, the BSP node tree will always use polygon splitting, as it is this splitting procedure that allows us to remove the ambiguous draw order situation that can arise when alpha polygons intersect each other. As discussed in the accompanying textbook, by splitting polygons that span a node, we break them into smaller pieces whose correct draw order can always be found during tree traversal.

As we are essentially building a clipped tree, we will use the same polygon storage logic used by our previous tree types when compiling a clipped tree. That is, prior to the commencement of the build process, a temporary copy of the tree's polygon database will be made. This vector will contain a list of all the polygons originally registered with the tree. The list will then be emptied and the recursive compilation process will be performed with the temporary copy. This will allow us to clip the polygons during the tree and only add them back to the tree's polygon database when they are assigned to a node in their clipped format. At the end of the build process, the tree's polygon database will contain all the polygons in the tree in their clipped format.

Every node in the tree will also be assigned an axis aligned bounding box that bounds not only the polygons stored at that node, but all the polygons in the nodes underneath it. This will allow us to construct a hierarchy of bounding boxes in our tree that can be used during the rendering traversal to reject nodes (and all their children) that lay outside the frustum. This will allow us to avoid visiting every node in the tree and make the rendering process much more efficient in the general case.

At the end of the CBSPNodeTree::Build function, prior to it returning program flow back to the application, it will issue a call to the CBSPNodeTree::PostBuild function. This is another process that we are familiar with and it will mirror the flow of our previously developed spatial trees. That is, it will call the CBSPNodeTree::CalculatePolyBounds method to calculate a bounding box for every polygon in the tree and then call the CBSPNodeTree::BuildRenderData method to copy the vertices of each CPolygon in the tree into a vertex buffer (building support structures to aid in the rendering of that data). Fortunately, the render system will not be nearly as complex as the system used to prepare and render our other spatial trees. After the BuildRenderData method returns, program flow goes back to the application and everything is ready to go. The CBSPNodeTree::Draw method can be used at any time thereafter to render the tree in a back to front order.

Finally, it is worth noting that at the moment, we are only using this tree for rendering. Thus, after the render data has been built and we have copied our vertex data into a vertex buffer, we could delete all of the CPolygon structures as we will not use them again. However, we decided to leave CPolygon structures at the node in case you ever do decide to perform some form of intersection testing on the BSP tree at a later date. We can certainly foresee situations in the future where we might want to read back the polygon vertex data stored in the tree and in that instance, we would want to have these system memory representations. However, in this lab project we do not use the CPolygons at all once the tree has been

built, so you are free to delete them after the BuildRenderData method has returned. Considering that the average game level probably only has a handful of transparent polygons (relatively speaking of course), leaving them in place in case we need them later will not increase them memory footprint of the application by any significant degree. The choice is yours.

The Render System

The rendering system for the BSP node tree will be very different from the rendering system used by CBaseTree and will require much less code. Our rendering system uses the following components:

- **The Vertex Buffer**
  The tree will have a single static vertex buffer which will be populated with the vertices of all the CPolygon structures during the BuildRenderData call. As the tree only supports a single vertex buffer this does mean that it is limited to managing a maximum of 65,535 vertices. While it may seem as if we should account for the multiple vertex buffer case, if we have 65,535 vertices whose polygons have to be rendered back to front each frame (and possibly one polygon at a time in the worst case scenario), we have bigger design problems than the need to update the code to support multiple vertex buffers. Essentially, we would have a huge tree with thousands of nodes which would slow traversal down quite a bit. Having to render this number of polygons in a back to front order (even if a BSP tree was not used and the polygons were sorted) is pretty much out of the question if we wish our frame rates to remain interactive.

- **The Dynamic Index Buffer**
  Our tree class will allocate a single dynamic index buffer that will be filled and flushed each time the tree is rendered. The BuildRenderData method will allocate this index buffer to be of the correct size, but will not initially store any data in it. The index buffer will be filled only during the recursive build procedure. As we visit each node in back to front order, we will copy the indices of each polygon stored in that node into the index buffer. If the triangles we are about to add do not have the same subset ID as those already in the index buffer (collected from prior visited nodes), the index buffer will be rendered and reset before the new polygon is added, thus starting a new collection run. This rendering logic will allow us to store multiple triangles across consecutively visited nodes in the index buffer together so that we can render them with a single draw call. We can imagine for example that if every alpha polygon in the tree had the same attribute ID (which is not so unreasonable, considering you might use a single effect for all of your glass/windows) we would actually collect the triangle data from all visited nodes into the index buffer and be able to render the entire tree dataset with a single draw call. Because we are visiting the nodes in a back to front order, the triangle will be added to the index buffer in that same order. Thus the triangles that should be rendered first are at the start of the index buffer and will be rendered first when the index buffer is submitted for rendering. As this index buffer will need to be locked and filled frequently we will need to create it with the D3DUSAGE_DYNAMIC flag so that the driver allows us to avoid stalling the pipeline.

- **Per-Polygon Index Data**
  We have already established how the rendering logic for the BSP tree should work. Every time the tree's draw function is called, a back to front traversal is done where every visible node add its

indices to a dynamic index buffer which is flushed whenever an attribute state change is encountered. The problem we have with this system however is that our CPolygon structures store convex winding N-gons and do not even store indices. Even if each CPolygon structure did store an array of indices, they would not be a lot of use to us as they would be local to the vertex array for that polygon instead of indexing into our main tree vertex buffer. This means that during the build process we are going to have to create a small index array for each polygon which will be stored in the node. So, if a node contains 5 polygons, we will also need to generate and store the corresponding 5 index arrays (one for each polygon) at that node. The index array for each polygon will describe the indices needed to render the triangles using vertices stored in the tree's vertex buffer.

The index data for each polygon at a node will be stored in a BSPRenderData structure (shown below). Note that this structure is actually contained inside the CBSPNode namespace but is shown here on its own. There will be one of these structures for each polygon stored in a node. As you can see, it contains an unsigned long pointer which will be used to point at the indices we generate for it, the triangle count that the polygon was broken into (which tells us how many triangles are in the previously described index array), and the attribute ID of the polygon that this index list is for. As we visit each node during the traversal pass, it is the indices stored in these structures that we will copy into our dynamic index buffer for eventual rendering.

```
struct BSPRenderData
 {
        USHORT * Indices;
        USHORT   TriCount;
        ULONG    AttribID;
 };
```

As there will be one BSPRenderData structure for each CPolygon stored at the node, one might well wonder why we are also storing the triangle attribute IDs in this structure when we can get the attribute ID from the corresponding polygon. After all, if there is one of these structures for each polygon stored at the node, then surely the attributes of the fourth polygon (for example) would be pNode->BSPRenderData[3].AttribID = pNode->Polygons[3].AttribID. That is, we should know that the attribute ID of the triangles stored in a BSPRenderData structure in the node's BSPRenderData array will match the attribute ID of the polygon stored in the node's polygon array at the corresponding element, right? In fact, this is not the case.

Although there will be a BSPRenderData structure allocated for each polygon at the node, we will make a small optimization in order to increase the chance of batch rendering multiple polygons stored at the same node with the same attribute ID. We will store the BSPRenderData structures in the array grouped together by subset so that when the node is encountered during the rendering traversal, we add all the polygons with the same attribute ID to that buffer first and then flush it before adding the second subset of polygons store at that node. To understand this, imagine the following polygons are all co-planar and were therefore assigned to the same node by the BSP tree compilation process. When this node is visited, the polygons A through F would have their indices collected into the dynamic index buffer and rendered.

The Node's CPolygon Array			The Node's BSPRenderData Array	
CPolygon A	:	AttribID 1	Indices A :	AttribID 1
CPolygon B	:	AttribID 2	Indices B :	AttribID 2
CPolygon C	:	AttribID 1	Indices C :	AttribID 1
CPolygon D	:	AttribID 2	Indices D :	AttribID 2
CPolygon E	:	AttribID 1	Indices E :	AttribID 1
CPolygon F	:	AttribID 2	Indices F :	AttribID 2

The above table shows how the CPolygon array stored at each node might have a 1:1 mapping with the BSPNodeRenderData array stored at that same node. That is, the first element in the BSPRenderData array stores the indices for the first element in the node's CPolygon array, and so on down the list. As we can see, it would seem pointless in this instance to store the attribute in both structures. However, look at the list of attributes in the above table and notice that the polygons that belong to same subset are not grouped at the node. We can see for example that we would first add the indices for polygon A (Indices A) to the dynamic index buffer. However, when we tried to add polygon B's indices, we would notice that the subset is different and would have to render the dynamic index buffer first before adding the second polygon. Because the polygons in this list are not stored in attribute order, and because we have to flush the index buffer every time a subset change is encountered, this would mean that although only two subsets of polygons exist at this node, every polygon would have to be rendered individually. This is extremely inefficient as it results in six draw calls. Of course, if we were to reshuffle the index arrays in the BSPRenderData array so that they are now grouped in attribute order, we would be able to render all six polygons with just two draw calls (one for each subset). Therefore, when we allocate and store the BSPRenderData structure for each CPolygon stored at a node, we will do so in subset order so that the BSPRenderData structure is essentially attribute sorted as shown below.

The Node's CPolygon Array			The Node's BSPRenderData Array	
CPolygon A	:	AttribID 1	Indices A :	AttribID 1
CPolygon B	:	AttribID 2	Indices C :	AttribID 1
CPolygon C	:	AttribID 1	Indices E :	AttribID 1
CPolygon D	:	AttribID 2	Indices B :	AttribID 2
CPolygon E	:	AttribID 1	Indices D :	AttribID 2
CPolygon F	:	AttribID 2	Indices F :	AttribID 2

Because we add the BSPRenderData structures to the array in subset order, this means there will not be a 1:1 mapping between polygons in the CPolygon array stored at that node and the index arrays in the BSPRenderData array. For example, we can see that polygon B is stored in the CPolygon array in the second position, but its indices are stored in the fourth element of the BSPRenderData array.

Looking at the second table above we can see that when this node is visited during the render traversal and the indices of each polygon are copied into the index buffer, we add the indices for polygons A, C, and B before we encounter a subset change and flush. Thus we can render these three polygons in a single draw call. We then collect the polygons B, D, and F into the index buffer and flush again the next time a subset change is encountered. We can see that when the

traversal function reaches a node, it will loop through its BSPRenderData array and copy the index arrays stored here into the dynamic index buffer, only flushing on a subset boundary. Of course, as we test and add each index array, we must know what attribute the polygon it pertains to has been assigned to. This lets us know if we can add it or if we have to flush the buffer before we do so. As the BSPRenderData array is now totally out of sync with the polygon array, we can no longer fetch it easily. That is why we also store the attribute ID in each BSPRenderData structure. It allows us to quickly detect a subset change when collecting the index arrays. Consequently, we know what subset has to be set up prior to flushing the index buffer.

- **The Attribute Callback**
  Another problem we have to overcome is that the BSP node tree essentially has to render its polygons in a self-managed fashion. With our previous tree types, we could allow the application to render the spatial tree's data batched by subset. The application would typically set the texture and material for a give subset and would then call the spatial tree's DrawSubset method to instruct the tree to render all the triangles collected into the associated render bin on the last visibility pass. The application cannot employ this same strategy when rendering the BSP node tree since the idea of rendering the scene by subset is mutually exclusive to the idea of a required back to front order, regardless of render state. Therefore, once the application calls the CBSPNodeTree::Draw function, it will not return until all of the polygons in the tree have been rendered. This leads to a very important question… where will the scene be able to set the texture and material for polygon (or batch of polygons) as it is rendered?

  The application will be able to register a callback function with the BSP tree that will be called prior to rendering a batch of triangles. That is, every time the index buffer needs to be flushed during the back to front render traversal, the registered callback function will be called by the tree and passed the attribute ID of all the triangles currently collected into the index buffer that are about to be rendered. The callback function will be responsible for setting the correct texture and material for the passed ID since the tree will have no idea what an attribute ID actually means.

  In our applications, it is the CScene class that has knowledge of what an attribute ID means (i.e., which texture and material it pertains to in the scene's attribute array). This is especially true in our application where the CScene class issues global subset IDs to all actors and meshes that are loaded. It makes sense then that is will be CScene class that will own the attribute callback function that will be registered with the BSP tree and called by that tree prior to rendering any data.

- **Lost/Reset Device Recovery**
  Because we have to use a dynamic index buffer for efficient triangle collection during the render traversal, we have an additional mechanism to implement in this tree that will actually force us to add a few small functions to CScene which will be called by CGameApp whenever the device is lost or reset.

  Up until now we have allocated all of our resources (such as index buffers, vertex buffers, and texture surfaces) in the managed resource pool (D3DPOOL_MANAGED). As we learned in Module I of this series, when we allocate a managed resource, we are allowing the Direct3D memory manager to manage our resource memory, leaving us with quite literally no work to do

after the resource has been created (other than releasing its interface when we no longer need it). The Direct3D memory manager will automatically create a system memory version of the resource data in addition to the data that may have been uploaded into video memory. This way, if the device is lost and then reset, the memory manager will automatically destroy the video memory resource and recreate and repopulate that resource with the copy of the data it has stored in the system memory version. As we know, system memory resources do not expire when the device becomes lost.

Since we now wish to create a dynamic index buffer, we will have to find another way to go because we are not allowed to create a dynamic resource in the managed resource pool (see Module I). So we will have to substitute the D3DPOOL_DEFAULT flag for the D3DPOOL_MANAGED flag during buffer allocation and use the default pool. If our device is lost, our BSP tree will have to release its index buffer and, when the device is reset, create a new one.

Destroying the index buffer and recreating it in the lost/reset device states is no hardship. Whenever the device is lost, CScene will call the CBSPNodeTree::PreReset method. This method will simply release the index buffer interface, destroying it and removing it from memory. When the device is eventually reset and ready for use by the application again, CScene will issue a call to the CBSPNodeTree::PostReset method which will simply recreate the dynamic index buffer. The problem we have with our framework so far is that the CScene class has no way of knowing when a device is in a lost state. This situation is trapped in CGameApp both during the FrameAdvance method (before we render anything) and when the user chooses to resize the window. Conversely, the CGameApp class also has no knowledge of the tree, since it is owned by the scene. Now we will need to add two new methods to CScene which can be called by CGameApp whenever it encounters a lost/reset device situation. The CGameApp class will issue a call to the CScene::PreReset method after the device has been found to be in a lost state and about to be reset. Once the device has been reset, CGameApp will call the CScene::PostReset method. These two new CScene methods simply give the scene a chance to forward the messages on to the objects it owns that need to be informed of such situations (the BSP tree as one obvious example). If we add new object types to CScene in the future that use default pool resources, we can simply let the scene call that object's PreReset and PostReset method from these two new functions.

We now have a fairly good high level understanding of the system that will be employed in order to support BSP node tree alpha rendering. From the application's perspective, the changes will be very light, as we will see shortly.

It is now time to examine the new code and any source code modifications that will be made to our demo framework. We will start by looking at the changes we will need to make to CGameApp and CScene before we study the code to the BSP node tree. This way, we can at least get an understanding of how it is ultimately going to be used in practice. Once done, we will examine the code to the node BSP tree itself.

Source Code Walkthrough - CGameApp

We will begin by examining the very small code changes that will need to be made to CGameApp in order to notify CScene about device losses and resets. No new functions or members have been added to CGameApp, but we have added two new function calls to CScene in the places where device reset detection is performed. Remember that we will generally have to reset the device when the window is resized or when it is restored after having been minimized (which always causes a lost device). Therefore, we will see calls to CScene::PreReset and CScene::PostReset inside the CGameApp::DisplayWndProc method that handles these messages. The same two functions will also be called inside the CGameApp::FrameAdvance method if it detects that the device was lost, has been recovered, and should therefore be reset. As the code added in both places is the same, we will look at only the modifications that have been made to the CGameApp::FrameAdvance method.

CGameApp::FrameAdvance (Modified)

The changes to this function are very minor, but we have shown the entire function again below (with most comments removed to compact the listing) so that we can get a feel for where the new functions are called.

The first part of the function is unchanged from previous lab projects. We increment the application counter first. Recall that the application counter is used by our spatial trees as a frame counter so that we do not test the same polygon twice during intersection tests if we are using a non-clipped spatial tree.

```
void CGameApp::FrameAdvance()
{
    static TCHAR FrameRate[ 50 ];
    static TCHAR TitleBuffer[ 255 ];

    // Increment our application counter
    IncrementAppCounter();
```

We then increment the application's timer using its (Tick method) and retrieve and print the current frame rate as calculated by the timer. We then display the current frame rate by setting it as the caption text of the render window.

```
    // Advance the timer
    m_Timer.Tick( );

    // Skip if app is inactive
    if ( !m_bActive ) return;

    if ( GetAppCounter() < 30 )
        m_Timer.SetSimulationSpeed( 0.0f );
    else
        m_Timer.SetSimulationSpeed( 1.0f );

    // Get / Display the framerate
    if ( m_LastFrameRate != m_Timer.GetFrameRate() )
    {
        m_LastFrameRate = m_Timer.GetFrameRate( FrameRate );
        _stprintf( TitleBuffer, _T("Collision Detection : %s"), FrameRate );
        SetWindowText( m_hWnd, TitleBuffer );
```

```
    } // End if Frame Rate Altered
```

This next section of code is modified. If the device is currently in a lost state and the IDirect3DDevice9::TestCooperativeLevel method is not returning D3DERR_DEVICENOTRESET, then it means the device is still lost and we can do nothing other than return. However, if we do get back an error code of D3DERR_DEVICENOTRESET then it means the device has just been recovered from a lost state and now needs to be reset before it can be used. Notice that before we reset the device (via the CD3DInitialize::Reset method -- see Chapter 2 of Module I) we first call the CScene::PreReset method. This will allow the scene to instruct the BSP node tree that it should release its index buffer since it is no longer valid. We then reset the device. After the call to the CD3DInitialize method returns, the device will have been fully recovered and non-managed resources will need to be rebuilt. We then call the CScene::PostReset method so that the scene can instruct the BSP tree that the device is now ready for rendering again and as such, it should re-create its index buffer.

```
    // Recover lost device if required
    if ( m_bLostDevice )
    {
        // Can we reset the device yet ?
        HRESULT hRet = m_pD3DDevice->TestCooperativeLevel();
        if ( hRet == D3DERR_DEVICENOTRESET )
        {
            // Restore the device
            CMyD3DInit Initialize;

            m_Scene.PreReset();

            Initialize.ResetDisplay( m_pD3DDevice, m_D3DSettings, m_hWnd );

            m_Scene.PostReset();

            SetupRenderStates( );
            m_bLostDevice = false;

        } // End if can reset
        else
        {
            return;

        } // End if cannot reset

    } // End if Device Lost
```

If the function has not returned at this point, it means we have a valid device to work with and we call all of our usual methods to fetch user input, update the player, update our animations, and render that scene. The function finally ends with the frame buffer being presented to the user.

```
  ProcessInput();
  m_Player.Update( m_Timer.GetTimeElapsed() );
  m_Scene.AnimateObjects( m_Timer );
  m_pCamera->UpdateRenderView( m_pD3DDevice );

  m_pD3DDevice->Clear( 0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0x797979, 1.0f, 0 );

  m_pD3DDevice->BeginScene();
  m_Scene.Render( *m_pCamera );
```

```
  m_pD3DDevice->EndScene();

  // Present the buffer
  if ( FAILED(m_pD3DDevice->Present( NULL, NULL, NULL, NULL )) ) m_bLostDevice = true;
}
```

Source Code Walkthrough - CScene (Modified)

As discussed in the previous section, two new methods will be added to CScene that will be called by CGameApp whenever the device has been recovered from a lost state and is about to be reset. These two new CScene methods are defined in CScene.h, as shown below.

Excerpt from CScene.h

```
void                PreReset        ( );
void                PostReset       ( );
```

We have also added a new member variable to CScene, which is a pointer to a CBSPNodeTree object. This is the object that manages our BSP node tree implementation. It is worth noting that because the BSP tree is only being used to store and render our transparent polygons, the scene will still have an ISpatialTree member just as it did in previous projects. This will make it clear that CBSPNodeTree is not a replacement for ISpatialTree derived spatial managers. Thus, below we see the two member variables in CScene that point to its tree objects.

Excerpt from CScene.h

```
ISpatialTree       *m_pSpatialTree;
CBSPNodeTree       *m_pAlphaTree;
```

The ISpatialTree derived object will still be used to contain all scene geometry for collision detection and will render all opaque geometry. The BSP tree will contain only alpha polygons and will be rendered in CScene::Render only after all other objects have been rendered (including all subsets of the ISpatialTree object).

We will now discuss the changes and additions to CScene so that we get a feel for where the BSP tree is created, where it is populated with polygon data, where it is compiled, and where and how it is rendered.

CScene::LoadSceneFromIWF (Modified)

The first function we will look at is the very familiar CScene::LoadSceneFromIWF. This is the top level function that encapsulates the entire scene loading process with respect to IWF file data. It is in this function that our ISpatialTree and CBSPNodeTree are first allocated and, at the bottom of this function, instructed to compile their polygon data that was registered during the IWF extraction process.

Since this function is getting a little lengthy, and only a few very small additions have been made to it, we will replace large sections of the function with the '….SNIP….' placeholder and a short comment describing the code we have decided not to show here in order to compact the listing on the printed

page. If you are following along with the lab project source code open in front of you (always a good idea), you will be able to see the full code to the function.

The first thing this function does is append the data path used by the application to the passed file so that the IWF file is loaded from the application's Data folder. That code has been in this section since the early days of Module I, so we will snip it from this listing. Notice that after we have constructed the full filename, we also clear the scene's collision object so that it contains no geometry or dynamic objects that might be left over from a previously loaded scene.

```
bool CScene::LoadSceneFromIWF( TCHAR * strFileName,
                               ULONG LightLimit /* = 0 */,
                               ULONG LightReservedCount /* = 0 */ )
{
    CFileIWF File;
    HRESULT  hRet;
    ULONG    i;

    ……SNIP……Append application data path to passed file name here

    // Clear our collision database
    m_Collision.Clear();
```

In this next section, which is executed before any data is loaded, we allocate both the spatial tree and the BSP node tree (called m_pAlphaTree) that will be used by our application. In this example the spatial tree allocated is of type CQuadTreeYV, but this can vary depending on the spatial tree you wish to use in your own application. The allocation of the spatial tree was here in the last two lessons as well, so all that has changed in this function is the addition of the one line that creates a new BSP node tree.

```
    // File loading may throw an exception
    try
    {
        // Allocate our spatial partitioning tree of the required type
        m_pSpatialTree = new CQuadTreeYV( m_pD3DDevice, m_bHardwareTnL, 200.0f, 100, 20 );
        m_pAlphaTree   = new CBSPNodeTree( m_pD3DDevice, m_bHardwareTnL );

        // Add our scene callback to the player.
        GetGameApp()->GetPlayer()->AddPlayerCallback( CScene::UpdatePlayer, this );
```

Notice at the bottom of the above listing that there is another familiar line. This is the call to the CGameApp owned player's AddPlayerCallback method, where the CScene::UpdatePlayer method is registered. It is this callback function that is invoked by CPlayer whenever the player is updated (this is the function that we added in our lessons on collision detection).

The next section also involves nothing new. We use the CFileIWF::Load method to load the IWF file data into its internal vectors and call a series of CScene::ProcessXX methods to extract the data and convert it into data structures useful to the application. The calls to the processing methods have been snipped as we have seen them many times before.

```
        // Attempt to load the file
        File.Load( strFileName );


        ……SNIP……Call the CScene::Process…… methods here to extract IWF data
```

```
        // Allow file loader to release any active objects
        File.ClearObjects();
```

At this point in the code both the spatial tree and the BSP tree will have had all scene polygons registered with them. This is because the CScene::ProcessMeshes is called from the above section of code (not shown) to extract the static mesh data from the IWF file. We saw in the previous lesson that for every face that we extract, we call CScene::ProcessVertices to allow our application to store the face data. As you will see in a moment, it is in this function that we create a new CPolygon structure from the passed vertices. We then test the face properties to see if it is an alpha face. If it is not an alpha face then it is simply added to the spatial tree as normal. If it is an alpha polygon, it is still registered with the spatial tree, but with the polygon's visibility Boolean set to false. This means that the polygon will be registered with the spatial tree, but will not be compiled into the render data version of the geometry. Therefore, collision queries made on the tree will correctly detect intersections with the alpha polygons, but when the tree was built, the vertices and indices of these alpha polygons would not have been added to the render data, so the polygons will not be rendered.

Of course, this is exactly what we want. We do not want the spatial tree to render our alpha polygons since they need to be rendered last and in a back to front order. Therefore, when we add an alpha polygon to the spatial tree with its visibility Booleans set to false, we make a copy of the polygon and add it to the BSP tree as well.

At this point in the LoadSceneFromIWF function then, every static polygon loaded from the IWF file will have been registered with the spatial tree and will be awaiting the build process. The BSP tree will also have had any static alpha polygons loaded from the file registered with it and is also awaiting the build process.

If the next section we set up any default lights if no lights were loaded from the file (otherwise we would not be able to see anything that we render) and instruct the spatial tree to build itself. After the build function returns, the tree will be built with all of the static polygon data loaded from the file, but its render data will contain only the opaque polygons. We then set the spatial tree as the collision system's broad phase test mechanism.

```
        // If no lights were loaded, lets default some half-way decent ones
        if ( m_nLightCount == 0 )
        {
            ……SNIP……Setup default lights here

        } // End if no lights

        // Build the spatial tree and notify the collision engine
        if ( !m_pSpatialTree->Build( ) ) return false;
        m_Collision.SetSpatialTree( m_pSpatialTree );
```

Next we instruct the BSP tree to build itself. After the compilation of this tree is complete, we set the attribute callback for later use during rendering.

```
        // Build alpha tree
        if ( !m_pAlphaTree->Build( ) ) return false;
        m_pAlphaTree->SetAttributeCallback( SetAttributes, this );
```

Notice in the above code that we set the alpha tree's attribute callback function to a CScene function called SetAttributes. We also assigned the CScene instance's pointer as the context data so that when this callback is triggered, it can use the 'this' pointer to access the CScene object's non-static data (such as its array of global attributes). The CScene::SetAttributes method will be called by the BSP tree whenever it is about to flush the index buffer and render a batch of triangles with matching attribute IDs. This function should set the correct attributes (texture and material) on the device so that the BSP tree can then render the triangles. We will look at the CScene::SetAttributes method in a moment.

In the next and final section of the code, we loop through each CTerrain object which may have been created during the load process and call the CTerrain::TreeBuildComplete. As we saw in the previous lesson, this function allows each terrain object to register each of its terrain blocks as a detail/dynamic object with the spatial tree. We then optimize the collision system, which essentially just removes any excess wasted memory that may currently exist at the end of its arrays/vectors, and allocate and initialize the CSoundManager class (which we will snip from the listing as it has not changed since it was first introduced several lessons ago).

```
        // Notify terrain that objects the spatial tree has been built
        for ( i = 0; i < m_nTerrainCount; ++i ) m_pTerrain[i]->TreeBuildComplete();

        // Optimize our collision database
        m_Collision.Optimize();

        ……SNIP……Setup CSound manager here

    } // End Try Block

    // Catch any exceptions
    catch (...)
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

By the time this function returns, both of the scene's trees have been compiled and are ready for use.

CScene::SetAttributes (New)

Although this method has been has been introduced in this lesson, it really is a handy function to have around for any future objects we implement which may need to perform their own rendering with no understanding of attribute IDs. As we know, the only object that understands what the attribute ID assigned to a triangle means is the CScene class, which stores the texture/material combinations in its m_pAttribCombo array when it is requested to load an attribute. As we saw back in Chapter 8, it is the index of the texture/material in the scene's array that is actually returned as the global attribute ID.
This function has a very simple task. It will be called by the BSP node tree (or any object that may wish to use it in the future) and will be passed a void pointer to context data and an attribute ID. As we saw above, the context data registered with the callback is the instance of the CScene object that registered

the callback. When the BSP tree needs to render some triangles, it will call this callback function, passing in the context pointer and the attribute ID of the triangles it currently has collected in its index buffer. This function can then cast the context pointer to a CScene object pointer which will then give us access to the non-static data. Thus, with this pointer, the attribute array can be accessed and the correct texture and material retrieved, as shown below.

```
void CScene::SetAttributes( LPVOID pContext, ULONG nAttribID )
{
    CScene * pScene = (CScene*)pContext;

    // Retrieve indices
    long MaterialIndex = pScene->m_pAttribCombo[nAttribID].MaterialIndex;
    long TextureIndex  = pScene->m_pAttribCombo[nAttribID].TextureIndex;
```

You will recall that the scene's m_pAttribCombo array is an array of ATTRIBUTE_ITEM structures, where each item describes a texture and material combination by specifying two array indices. The first is the index of a texture in the scene's texture array and the second is the index of a material in the scene's material array. The index of an ATTRIBUTE_ITEM structure in this array also describes the attribute ID of that combination. So, in the above code you can see that we use the attribute ID to fetch the material and texture indices from the ATTRIBUTE_ITEM structure for the passed attribute ID.

Now that we have the index of the texture and material in the scene's respective arrays, we fetch the texture and material and bind them to the device. If the material index is less than zero, then it means there is no material assigned to this attribute and that the scene should use its default material.

```
    // Set the states
    if ( MaterialIndex >= 0 )
        pScene->m_pD3DDevice->SetMaterial( &pScene->m_pMaterialList[ MaterialIndex ] );
    else
        pScene->m_pD3DDevice->SetMaterial( &pScene->m_DefaultMaterial );
```

If the texture index is less than zero, then we will set NULL for the texture in texture stage 0 since it means the passed attribute has no texture assigned. If it is a valid index, we fetch the relevant element from the scene's TEXTURE_ITEM array and bind the texture pointer stored there to texture stage 0.

```
    if ( TextureIndex >= 0 && pScene->m_pTextureList[ TextureIndex ] )
        pScene->m_pD3DDevice->SetTexture( 0,
                                          pScene->m_pTextureList[ TextureIndex ]->Texture );
    else
        pScene->m_pD3DDevice->SetTexture( 0, NULL );

}
```

Although this is a simple function that requests that the scene set up the device attributes specified by the caller, we can certainly imagine how it may be used in the future by other components, not just by our BSP node tree. It is quite possible that you will develop other components down the road that for some reason or another have to render their polygons just like our BSP node tree. These components can now make a call to this function before rendering a given subset of triangles so that the scene can make sure that the correct attributes are mapped to the device prior to the commencement of rendering.

CScene::ProcessVertices (Modified)

This function has also been part of our framework since the early lessons of Module I. It is called from ProcessMeshes when a polygon needs to be processed and stored by the application. You will also recall that in the last lesson, this function was modified so that the vertices passed to this function are now used to construct a CPolygon structure that is registered with the scene's spatial tree. Now we will also add an additional piece of code that tests to see if the passed polygon is an alpha surface. If so, it will be added to the spatial tree as an invisible polygon and then copied and added to the BSP tree. The BSP tree will be the one responsible for rendering it. We will step through the first part of the function quickly as we have seen it many times before and it is unchanged from previous versions.

The function is passed the polygon as an iwfSurface pointer along with its attribute ID. The first thing we do is allocate a new CPolygon structure which we will use to store this new data. We set the attribute ID of the polygon to the attribute ID passed into the function and copy over the face normal from the passed iwfSurface. If the BackFace parameter to this function has been set to true by the caller then it means that this vertex needs its winding order flipped so that it is facing in the opposite direction. We should also negate the normal in this case as well.

```
bool CScene::ProcessVertices( iwfSurface * pFilePoly,
                              ULONG        nAttribID,
                              bool         BackFace /* = false */ )
{
    // Validate parameters
    if ( !pFilePoly ) return false;

    long      i, nOffset = 0;
    CVertex * pVertices = NULL;
    float     fScale = 1.00f;

    // Allocate a new empty polygon
    CPolygon * pPolygon = new CPolygon;
    if ( !pPolygon ) return false;

    // Set the polygon's attribute ID
    pPolygon->m_nAttribID = nAttribID;
    pPolygon->m_vecNormal = (D3DXVECTOR3&)pFilePoly->Normal;

    if ( BackFace ) pPolygon->m_vecNormal = -pPolygon->m_vecNormal;
```

In the next section we fetch the number of vertices contained in the passed iwfSurface structure and pass this value into the CPolygon::AddVertex method which will allocate the CPolygon's array of vertices to the correct size. We then fetch a pointer to the polygon's vertex array so that we can later use it to fill the vertex array with meaningful data.

```
    // Allocate enough vertices
    if ( pPolygon->AddVertex( pFilePoly->VertexCount ) < 0 ) return false;

    pVertices = pPolygon->m_pVertex;
```

In the next section we will loop through each vertex in the iwfSurface structure and copy its information into the vertex array. In the case of a normal polygon this is a simple loop that steps from 0 to N, where N is the last vertex. However, if the BackFace parameter has been set to true then it means the caller wants the passed polygon to face in the opposite direction. We have already negated the polygon normal, but as we know, that is no enough; we must also reverse the winding order such that the vertices are added in the order N to 0. We do this all in one loop by using the nOffset variable. It will be set to 0 for normal polygons and N for back facing polygons. In each iteration of the loop we will access the vertex to copy from the iwfSurface using [i+offset] which equates to [i+0] in the normal case and [i+N] in the back facing case. As *i* is increased with each iteration of the loop, nOffset is decremented by 2 (in the back face case only) such that, with each iteration of the loop [i+nOffset] will become one less. This allows us to step backwards through the winding order of the original surface and add the polygons in a counter-clockwise fashion.

Here is the code that copies over the vertices from the iwfSurface into the CPolygon's vertex array:

```
    // If we are adding a back-face, setup the offset
    if ( BackFace ) nOffset = pFilePoly->VertexCount - 1;

    // Loop through each vertex and copy required data.
    for ( i = 0; i < (signed)pFilePoly->VertexCount; i++ )
    {
        // Copy over vertex data
        pVertices[i + nOffset].x      = pFilePoly->Vertices[i].x * fScale;
        pVertices[i + nOffset].y      = pFilePoly->Vertices[i].y * fScale;
        pVertices[i + nOffset].z      = pFilePoly->Vertices[i].z * fScale;
        pVertices[i + nOffset].Normal = (D3DXVECTOR3&)pFilePoly->Vertices[i].Normal;
        if ( BackFace ) pVertices[i + nOffset].Normal = -pVertices[i + nOffset].Normal;

        // If we have any texture coordinates, set them
        if ( pFilePoly->TexChannelCount > 0 && pFilePoly->TexCoordSize[0] == 2 )
        {
            pVertices[i + nOffset].tu = pFilePoly->Vertices[i].TexCoords[0][0];
            pVertices[i + nOffset].tv = pFilePoly->Vertices[i].TexCoords[0][1];

        } // End if has tex coordinates

        // If we're adding the backface, decrement the offset
        // Remember, we decrement by two here because 'i' will increment
        if ( BackFace ) nOffset -= 2;

    } // Next Vertex
```

At this point our CPolygon has been populated with its vertex data (and texture coordinates if the passed surface had texture coordinates defined for it). We will now extract the source and destination blend modes that have been specified for this surface by the IWF creator.

> **Note:** In an IWF file, a 'Channel' can be thought of as a texture/material combination (sometimes just one of the two) and the settings describing how that channel should be rendered. If we think about a polygon that has been saved by the artist to use a blend of 3 textures, the surface would have three channels in the IWF file. Each channel describes the texture itself and any other settings required to render that texture/material correctly. If a surface has a channel count of 0, it means that it has no texture or material applied to it.

If the surface we have been passed has the SCOMPONENT_BLENDMODES flag set in its Components bitset, it means it has blend modes defined for it that can be found in the BLEND_MODE structure. This structure is defined in the file libIWF.h as shown below.

Excerpt from libIWF.h – Part of the IWF SDK

```
typedef struct _BLEND_MODE
{
    UCHAR       SrcBlendMode;               // Source Blend Mode
    UCHAR       DestBlendMode;              // Destination Blend Mode
} BLEND_MODE;
```

This structure contains two numeric values that describe the source blend and destination blend modes that should be used for this polygon. If the surface does not have this property defined, you can see at the top of the next section of code that we default the blend mode structure to contain zero in both its source and destination blend values. This basically tells us that no alpha blending is required.

```
    // Determine the blend modes we are using
    BLEND_MODE BlendMode = { 0, 0 };

    if ( (pFilePoly->Components & SCOMPONENT_BLENDMODES) && pFilePoly->ChannelCount > 0 )
          BlendMode = pFilePoly->BlendModes[0];
```

Notice that when we extract the blend modes from the surface in the above code, we extract from element zero in the surface's blend modes array. This is because the IWF file will store blend modes for each channel. Since we are not multi-texturing at the moment, we are only interested in fetching the blend modes for the first channel (the first and only texture).

Now that we have the blend modes for our polygon, we will test to see if either the source or destination blend modes are non-zero. If they are, it means that this surface should be alpha blended. In this case, we make a copy of the CPolygon (via a copy constructor) and register it with the alpha tree. The reason we create a copy of the CPolygon and do not just pass its pointer straight into the BSP tree is that we will also pass this polygon into spatial tree as well. We cannot use the same pointer for both trees because the BSP tree will most likely split the polygon during the build process, thus removing the original polygon are replacing it with two new child polygons. If the BSP tree deletes the polygon that also has a pointer stored in ISpatialTree, we are going to have significant problems. Thus, we make a copy and give each tree its own version to work with as it pleases.

Here is the code that copies the polygon and registers it with the alpha tree:

```
    // Is this using alpha blending?
    if ( BlendMode.DestBlendMode != 0 || BlendMode.SrcBlendMode !=0 )
    {
        // Make a COPY of the alpha polygon to add to the alpha tree
        CPolygon * pAlphaPoly = new CPolygon( pPolygon );
        if ( !pAlphaPoly ) { delete pPolygon; return false; }

        // Add this new polygon to the alpha tree
        if ( !m_pAlphaTree->AddPolygon( pAlphaPoly ) ) { delete pPolygon; return false; }

        // Ensure that when it's added to the spatial tree, that it's not used for rendering
        pPolygon->m_bVisible = false;
    }
```

Notice that after we added the copy of the polygon (pAlphaPoly) to the BSP tree, we set the visibility Boolean in the original polygon to false. This is because this polygon is about to be registered with the spatial tree and, although we want it added to the spatial tree for collision queries, we do not want the spatial tree to add this polygon to any of its vertex/ index buffers when it compiles its render data. This way, the polygon will not be rendered by the spatial tree and this task can be left to the BSP tree.

In the final section of code, we register the original polygon (which now has its visibility Boolean set to false) with the spatial tree, and we are done.

```
    // Add this new polygon to the spatial tree
    if ( !m_pSpatialTree->AddPolygon( pPolygon ) ) { delete pPolygon; return false; }

    // Success!
    return true;
}
```

During the loading process, this function will be called for every static polygon loaded from the IWF file. By ‘static’ we mean polygon data stored inside the IWF file (i.e., it is not called for polygons contained in external reference objects) which is defined in world space. All of these polygons will have been registered with the spatial tree and any surfaces that have been flagged by the artist as being alpha surfaces will be contained in the alpha (BSP node) tree. As we have seen, after all polygons have been processed, program flow returns to the CScene::LoadSceneFromIWF function where both trees are then instructed to compile their data.

CScene::PreReset (New)

As mentioned, the alpha tree will be using an index buffer that cannot be allocated in the managed resource pool. This means the resource will need to be rebuilt when the device is recovered from a lost state. The CGameApp object will call the CScene::PreReset method as soon as the device has been recovered, but has not yet been reset for use. This lets the scene inform any of its components to release any non-managed resources that are no longer valid. Currently, the only object in the scene that uses non-managed resources is the BSP node tree. As you can see, this function simply forwards the request to the BSP object’s function of the same name. In the BSP tree function, the dynamic index buffer will have its interface released.

```
void CScene::PreReset( )
{
    if (m_pAlphaTree) m_pAlphaTree->PreReset();
}
```

In the future we may wish to use other components that use non-managed resources and thus need to be notified of the same status. So it is a handy thing that we have now implemented a means for CGameApp to communicate the lost/reset device state to the scene.

CScene::Post Reset (New)

This function is called when the device has been reset after being recovered from a lost state. At the time this function is called, the device will have been reset and will be valid for use. This function passes the request on to any system components that need to rebuild their non-managed resources. In the case of the BSP node tree, its function of the same name creates a new dynamic index buffer which will be owned by the new recently reset device.

```
void CScene::PostReset( )
{
    if (m_pAlphaTree) m_pAlphaTree->PostReset();
}
```

We will look at the code to both the CBSPNodeTree::PreReset and CBSPNodeTree::PostReset methods in a moment when we cover the code to the BSP tree class.

CScene::Render (Modified)

The last modified function we need to look at in our framework is CScene::Render. All we have added to this function are two lines of code to render the BSP tree and set the device's world matrix to identity before doing so.

Below we show the CScene::Render function so that we can see the general flow and order in which things are done. However, as this function has hardly changed from previous versions, we have snipped out most of the code that renders the dynamic objects, sets up fog and lighting states, etc. Please consult the source code to Lab Project 16.1 if you would like to follow along with the discussion.

The first section sets up a few render states, renders the scene's sky box, and then sets any lights that the scene is currently using.

```
void CScene::Render( CCamera & Camera )
{
    ULONG i, j;
    long  MaterialIndex, TextureIndex;

    if ( !m_pD3DDevice ) return;

    ……SNIP……Set up render states

    // Render the skybox first !
    RenderSkyBox( Camera );

    ……SNIP……Set up lights

    ……SNIP……Set up fog
```

We next instruct the spatial tree to perform its visibility processing step. When this function returns, all leaves that exist inside the view frustum will be flagged as visible and any triangles that need to be rendered will be contained in their relevant leaf bins.

```
    // Allow the spatial tree to process visibility
    m_pSpatialTree->ProcessVisibility( Camera );
```

Our next task is to render the spatial tree leaf bins (one exists for each subset currently being used by the scene). As the static data contained in the spatial tree is already defined in world space, we will set the device world matrix to identity to prevent any additional transformation.

Below we see an abbreviated version of the code that loops through every subset currently being used by the scene, sets the texture and material associated with that subset (snipped), and then calls the ISpatialTree::DrawSubset method to render the polygons in the associated leaf bin.

```
    // Loop through each scene owned attribute
    D3DXMATRIX mtxIdentity;
    D3DXMatrixIdentity( &mtxIdentity );
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxIdentity );

    for ( j = 0; j < m_nAttribCount; j++ )
    {
        ……SNIP……Setup textures and materials on device for this subet

         m_pSpatialTree->DrawSubset( j );

    } // Next Attribute
```

With all the opaque static geometry rendered, we next loop through each CObject in the scene's CObject array and render all the subsets of any mesh/actor stored there. Once again we have snipped this section of the code as it has not been changed.

```
    // Process each object
    for ( i = 0; i < m_nObjectCount; ++i )
    {
        ……SNIP…… Render eacy dynamic object here (either actor or mesh)
    } // Next Object
```

The next section, which renders our terrain(s), is also shown using only placeholder code. Since our terrains are pre-lit in this demo, we disable lighting first, loop through each terrain, and call its Render method, before once again enabling the lighting pipeline.

```
    // Disable lighting for terrain
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, FALSE );

    // Render each terrain
    for ( i = 0; i < m_nTerrainCount; ++i ) m_pTerrain[i]->Render( &Camera );

    // Re-Enable lighting for alpha objects
    m_pD3DDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
```

In previous versions of this function the entire scene would now be rendered. Nothing more would be left to do other than call the ISpatialTree::DebugDraw method if the user has requested debug drawing of the spatial tree. However, we know that the last thing we render now should be our BSP tree. It will step through its nodes with the passed camera and render every visible polygon in a back to front order.

As the static data stored inside the BSP tree is already defined in world space, we first set the device's world matrix to identity before calling the CBSPNodeTree::Draw method.

```
    D3DXMatrixIdentity( &mtxIdentity );
    m_pD3DDevice->SetTransform( D3DTS_WORLD, &mtxIdentity );

    // Render alpha tree last
    m_pAlphaTree->Draw( Camera );

    ……SNIP……Disable Lights and Fog render states here

    // Draw the spatial tree's debug data if requested
    if ( GetGameApp()->GetDebugDraw() ) m_pSpatialTree->DebugDraw( Camera );

}
```

As you can see, after the BSP tree has been rendered we have finished rendering the entire scene. All that is left to do is disable any lighting and fog states that we used. As the final step in the function we call the ISpatialTree::DebugDraw method if the user has selected debug drawing from the menu.

We have now seen how the BSP node tree fits into our application and the few changes that were needed to accommodate it. Now that we have a feeling for how it works and where it is populated and rendered, let us have a look at the code to the CBSPNodeTree class and implement our first BSP compiler.

Source Code Walkthrough - CBSPNodeTree

Discussing the code to a BSP compiler would be fairly daunting were it not for the fact that in the previous lessons we covered a number of tree types. At this point, we are used to the recursive build process and the ways in which trees are constructed and traversed. Because of the material covered in the last two lessons, you will probably find that writing our first BSP compiler is relatively trivial. This is certainly true when you consider the extents we went to in order to devise an efficient rendering strategy for our spatial trees.

Building the BSP node tree is almost identical to building a clipped kD-tree. The main exception is that the split plane chosen at every node is selected from the polygon data that made it into that node and not calculated as one of the three axis aligned planes. Each node in the tree will have two pointers to child nodes that exist in the front and back halfspace of the node, just like the kD-tree. Of course, one other big difference with the node tree is that the polygons are stored at the nodes in the tree that they share a plane with. They are not collected at the terminal nodes (leaves), as with our previous spatial trees. Therefore, each node will also have the ability to store a list of polygons.

Before we look at the code to the CBSPNodeTree class, let us first look at the structure it uses to construct the tree and store the node information. The CBSPNode class is defined in CBSPNodeTree.h and is shown below, followed by an explanation of each of its member variables. Note that many of these members are similar to the members that we stored in the nodes/leaves of our other spatial trees.

Excerpt from CBSPNodeTree.h (CBSPNode class)

```
class CBSPNode
{
public:
    struct BSPRenderData
    {
        USHORT * Indices;
        USHORT   TriCount;
        ULONG    AttribID;
    };

    // Public Typedefs, structures and enumerators.
    typedef std::list<CPolygon*> PolygonList;


    // Constructors & Destructors for This Class.
     CBSPNode( );
    ~CBSPNode( );


    // Public Variables for This Class
    D3DXPLANE      Plane;              // Splitting plane for this node
    CBSPNode    *  Front;              // Node in front of the plane
    CBSPNode    *  Back;               // Node behind the plane
    PolygonList    Polygons;           // List of polygons assigned to this node
    BSPRenderData* PolygonData;        // One item per polygon
    D3DXVECTOR3    BoundsMin;          // Minimum bounding box extents
    D3DXVECTOR3    BoundsMax;          // Maximum bounding box extents
    signed char    LastFrustumPlane;   // The frame-to-frame coherance 'last plane' index.
};
```

D3DXPLANE Plane

This member will store the split plane that was selected when the node was constructed during the compilation process. The two child nodes will represent areas that are contained in each halfspace of this node.

When a node is first constructed during compilation, its plane is selected from the list of polygons that made it into that node. The polygon selected will be removed from the list so that it is not passed down to child nodes. Instead, it will be added to the polygon list for the node. The polygon vertices and normal will be extracted and used to create a D3DXPLANE structure representing the plane on which the polygon resides. The rest of the polygons in the list will then be classified against this plane and organized into front and back lists. Any polygons in the list that span the plane will be split and each child fragment added to the relevant list. If a polygon in the list is found to lie on the same plane, it will also be removed from the list and stored in the node's polygon list. Therefore, a node's polygon list will contain an array of all the planes whose vertices all reside on the node plane. The polygons' normals must also face into the front space of the plane.

CBSPNode * Front

This is the pointer to the node's front child. If it is not NULL, this points to a node that represents the frontspace of the current node.

CBSPNode * Back

This is the pointer to the node's back child. If it is not NULL, this points to a node that represents the backspace of the current node.

PolygonList Polygons
This is a list of polygons that are stored at the node. During the building process, any polygon that lies on the same plane as the node's split plane and faces in a matching direction will be added to this list. As the BSP tree uses plane depth to perform its back to front ordering, all polygons that lay on the same plane can be rendered together. This is why in the node tree case we store all co-planar polygons at the node. A node will always have a minimum of one polygon in its list, since a polygon was used to create the node plane. As such, at least that polygon will always be stored in this list. Typically, there are many polygons in a scene that share the same plane and in such situations, all of them would be assigned to a single node and rendered when the node is visited during the draw traversal.

BSPRenderData* PolygonData
As discussed earlier, we will be using a dynamic index buffer that is passed through the tree to collect triangles in a back to front order. For this system to work, the polygons in the tree must have indices available. Since our CPolygon structure does not store indices, this separate structure is used to store the indices of all triangles generated from a ***single*** polygon.

Excerpt from CBSPNodeTree.h (CBSPNode class)
```
struct BSPRenderData
{
       USHORT * Indices;
       USHORT   TriCount;
       ULONG    AttribID;
};
```

There will be one of these structures for each polygon stored in the node list. As you can see, each one stores an index array describing the triangles. There indices index into the main vertex buffer that is used by the tree to store the renderable version of the BSP geometry. Later we will see that after the tree has been compiled, the CBSPNodeTree::BuildRenderData function will be called. This function will copy all of the vertices in every polygon in every node into a vertex buffer and then traverse the tree building the BSPRenderData structure for each polygon at every node.

As discussed, although number of BSPRenderData structures stored at a node will naturally be equal to the number of polygons stored at that node, there is not a 1:1 mapping between the two arrays. This is due to the fact that we will create the BSPRenderData structures for each polygon stored in the node in attribute order. That is, the BSPRenderData array will be attribute sorted so that we can minimize the number of index buffer flushes that will have to be performed during the rendering traversal.

D3DXVECTOR3 BoundsMin
D3DXVECTOR3 BoundsMax
As with all of our previous tree types, each node will also store a bounding box that encompasses its polygons and the polygons stored underneath it in the tree. Although the box will generally not be nearly as good a fit as the box shaped nodes in our other spatial tree's, the end result is still a hierarchy of bounding boxes that can be used during the rendering traversal to perform hierarchical frustum culling.

signed char LastFrustumPlane
This is another member that was included in the previous lesson's nodes. It will store the index of the frustum plane that this node was rejected by in the previous traversal. This allows us to test this frustum

plane first during the next traversal so that it will hopefully be rejected again without having to test the five other frustum planes.

Now that we have an understanding of the information that will be stored at each node, let us finally begin examining the code to the BSP node tree. Our coverage will center on a class called CBSPNodeTree whose declaration can be found in CBSPNodeTree.h. We will look at the declaration one section at a time so that we can have a preliminary discussion about some of its methods and how they are used.

In order to make the node tree code feel more familiar, you will see that while the node tree is not derived from ISpatialTree (for the reasons discussed earlier), many of the methods that perform the same tasks have been given the same names. This way, we will already basically know at what point they are called in the build process and what tasks they are expected to perform. After the previous lessons, you should feel quite comfortable with most of these methods.

Excerpt from CBSPNodeTree.h

```
class CBSPNodeTree
{
public:


    // Constructors & Destructors for This Class.
    virtual ~CBSPNodeTree();
             CBSPNodeTree( LPDIRECT3DDEVICE9 pDevice, bool bHardwareTnL );


    // Public Functions for This Class.
    bool        Build               ( );
    bool        AddPolygon          ( CPolygon * pPolygon );
    void        SetAttributeCallback( void * pFunc, void * pContext );
    void        Draw                ( CCamera & Camera );

    void        PreReset            ( );
    void        PostReset           ( );
```

Above we see all the public methods exposed by this class. The Build method is called by the application after all polygon data has been registered with the tree. It is the top level function that starts the recursive build process. The AddPolygon function is called by the application prior to the tree being built to register polygon data with the tree. It is passed a pointer to a CPolygon structure that is stored in the tree's polygon vector. The SetAttributeCallback method is another method we have seen being used earlier when discussing the changes to CScene. This method is called so that the application can register a function pointer that the tree will call prior to rendering any triangles. This function will be expected to set the texture and material on the device before it returns so that when the tree flushes its dynamic index buffer, those triangles are rendered with the correct attributes. The Draw method we saw being called by the CScene::Render method. It is passed the camera that will be used to traverse the tree. It is the position of the camera which will determine the back to front traversal order. The PreReset and PostReset methods are called by the scene when the device has been lost and then restored. These two functions will be responsible for destroying the current index buffer and recreating it, respectively.

Now we will look at the private methods which will be invoked either directly or indirectly by the public methods discussed above.

Excerpt from CBSPNodeTree.h

```
private:

    // Private Typedefs, structures and enumerators.
       typedef std::list<CPolygon*> PolygonList;


    // Private Functions for This Class

    bool        BuildTree           ( CBSPNode * pNode, PolygonList PolyList );

    CPolygon *  SelectBestSplitter  ( PolygonList PolyList,
                                      unsigned long SplitterSample,
                                      float SplitHeuristic );

    void        DrawRecurse         ( CBSPNode * pNode,
                                      CCamera & Camera,
                                      UCHAR FrustumBits = 0x0,
                                      bool bAutoVisible = false );
```

The BuildTree method is called from the Build method and is the function that performs the recursive compilation of the tree. Each time the BuildTree method is called, it constructs and populates a node and then calls itself to create the children.

The SelectBestSplitter method will be called by the BuildTree method when it needs to select a polygon from the list passed into that node to be used as the split plane. As discussed in the accompanying textbook, this function will use an algorithm that will select a polygon that strives for the smallest number of splits while still factoring in tree balance as a key concern.

The DrawRecurse method is invoked by the Draw method to recursively walk the tree in a back to front order with respect to the passed camera position. Any node that is outside the frustum will not have its children visited and any node that is visited will have its polygons collected into the dynamic index buffer and rendered each time an attribute change is encountered.

Below we see the final set of private methods. Most of them should be recognizable by name, due to our coverage of CBaseTree.

```
    bool        Repair              ( );
    void        RepairTJunctions    ( CPolygon * pPoly1, CPolygon * pPoly2 );
    bool        PostBuild           ( );
    void        CalculatePolyBounds ( );
    bool        BuildRenderData     ( );
    bool        PopulateNodeData    ( CBSPNode * pNode, USHORT *& pIndices );
```

The PostBuild method is called from the Build method after the tree has been compiled. It first calls the CalculatePolyBounds method so that each CPolygon has a bounding box calculated for it. Although we never use the polygon bounding boxes in our lab project, it does not mean that you will not want to run intersection tests on the BSP node tree in your own application. In that case, the polygon bounding boxes are very useful to have to speed up these processes. After the CalculatePolyBounds methods returns, PostBuild then calls the BuildRenderData method to instruct the tree to compile the vertex data of the tree into vertex buffers, build the index array for each polygon at the nodes, and create a dynamic index buffer.

Before the BuildRenderData method starts to compile its data, it will call the Repair method to remove any T-junctions from the geometry that may have been introduced by the clipping process. This method calls the RepairTJunctions method as a helper function to mend the T junctions between a pair of polygons. These methods are just simplified versions of the repair methods that were in CBaseTree. As such, the code to these functions should be instantly understood.

After the BuildRenderData method has removed any T-junctions and collected all the vertices in the tree, it will then issue a call to the PopulateNodeData method, passing in the root node. This recursive method will traverse the tree and, at each node, build the index arrays for each polygon in attribute order.

Finally, now that we have a feel for the function layout and where in the process each will be called (most of which should be very familiar after the previous lesson), we will now discuss the member variables of CBSPNodeTree.

Excerpt from CBSPNodeTree.h

```
    // Private Variables for This Class
    CBSPNode               * m_pRootNode;       // The root node of the tree
    LPDIRECT3DDEVICE9        m_pD3DDevice;      // The Direct3D Device
    bool                     m_bHardwareTnL;    // Use hardware transformation and lighting?.
    LPDIRECT3DVERTEXBUFFER9  m_pVertexBuffer;   // Vertex buffer for storage of polygon data
    LPDIRECT3DINDEXBUFFER9   m_pIndexBuffer;    // Dynamic index buffer used for rendering.
    ULONG                    m_nVertexCount;    // Number of vertices stored in VB
    ULONG                    m_nFaceCount;      // Number of faces we need to store in IB.
    PolygonList              m_Polygons;        // LIST of Stored polygon data.
    CALLBACK_FUNC            m_AttribCallback;  // Callback function pointer
}
```

CBSPNode * m_pRootNode

After the tree has been compiled (i.e., after Build has been called by the application) this member will point to the root node of the BSP node tree.

LPDIRECT3DDEVICE9 m_pD3DDevice

This member will store a pointer to the device interface on which the BSP tree data will be rendered. This is the device that will own the vertex and index buffers for this tree.

bool m_bHardwareTnL

This Boolean signifies whether the device supports hardware accelerated transformation and lighting. If it is set to false, then software vertex processing will be needed and the tree will have to create its index and vertex buffers in system memory.

LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer

This is a pointer to the vertex buffer that will be created in the BuildRenderData method. It will contain all the vertex data for all triangles stored in the BSP tree.

LPDIRECT3DINDEXBUFFER9 m_pIndexBuffer

This is a pointer to the dynamic index buffer that will be created in the BuildRenderData call. This index buffer will not initially be filled with any data; it will be repeatedly filled and flushed during the rendering traversal.

ULONG m_nVertexCount;
This member describes the number of vertices in the tree's vertex buffer.

ULONG m_nFaceCount
This is the number of triangles in the entire tree. If we think of the vertex buffer as being a single mesh represented as an indexed triangle list, this value will be the number of triangles in that mesh. We obviously need to store this value and have it easily accessible whenever we need to create (or recreate in the case of a lost device) the index buffer since our index buffer must be large enough to collect this many vertices. Of course, it is probably rare that we would ever collect all of the triangles into the index buffer at once (considering frustum culling and buffer flushing on subset changes). However, it is still within the realm of possibility that the alpha polygons in this tree may belong to the same subset and may all be inside the frustum simultaneously. In such a situation, the render traversal would collect every single triangle stored in the tree and render the whole batch at once. Therefore, by allocating the index buffer large enough to contain every triangle, we can rest assured that we will always have enough room, regardless of the circumstances.

PolygonList m_Polygons
This is an STL list of CPolygon structures that will contain all the polygons currently registered with the tree (via the CBSPNodeTree::AddPolygon method). Pre-compilation, this list will contain all the polygon data that the scene has registered. Post-compilation, it will contain the data in its split format, just list the other tree types we developed with splitting enabled. In the BSP tree, not splitting is not an option. It is necessary for the algorithm to work correctly.

CALLBACK_FUNC m_AttribCallback
This structure has been used by several of our objects in the past to store callback functions and their context data. This structure contains two void pointers. The first stores a pointer to the callback function that was registered with the tree by the application, and the second is the context data that should be passed into the callback function when it is triggered. As usual, we will use the context data to pass the CScene object's pointer back to the scene's static callback function so that it has access to the non-static variables.

With all member variable discussions out of the way, let us start to study the functions to the BSP tree class in an order that most closely follows the flow of when they are called during the building and rendering process.

CBSPNodeTree::SetAttributeCallback

As we saw earlier, this method is called by CScene::LoadSceneFromIWF file just after the BSP tree has been created. It is passed a pointer to a static CScene function called SetAttributes which is called during rendering to set the texture and material for a given subset of triangles that the tree is about to draw. The second parameter passed to this function is the context pointer (the CScene instance that registered the callback) which will be passed to the callback function.

```
void CBSPNodeTree::SetAttributeCallback( void * pFunc, void * pContext )
{
    // Store details
```

```
    m_AttribCallback.pFunction = pFunc;
    m_AttribCallback.pContext  = pContext;
}
```

CBSPNodeTree::AddPolygon

This function is called by the application during the loading process when it wishes to register a polygon with the tree. As you can see, this function simply adds the passed CPolygon pointer to the back of the tree's m_Polygons STL list of polygons. It is this polygon list that will be the input data for the root node at the start of compilation process.

```
bool CBSPNodeTree::AddPolygon( CPolygon * pPolygon )
{
    try
    {
        // Add to the polygon list
        m_Polygons.push_back( pPolygon );

    } // End Try Block

    catch ( ... )
    {
        return false;

    } // End Catch Block

    // Success!
    return true;
}
```

CBSPNodeTree::Build

The Build function is called by the application to start the compilation process. The function first allocates the root node as shown below.

```
bool CBSPNodeTree::Build( )
{
    PolygonList::iterator       PolyIterator = m_Polygons.begin();
    PolygonList                 PolyList;

    // Nothing to build?
    if ( m_Polygons.size() == 0 ) return true;

    // Allocate a new root node
    m_pRootNode = new CBSPNode;
    if ( !m_pRootNode ) return false;
```

What it does next is not new to us as we employed the same strategy in all of the Build functions encountered for our other tree types when a splitting tree was being built. We make a copy of the tree's polygon list that currently contains all the registered polygons. The temporary list will be input into the recursive process and the original polygon list of the tree is emptied. This is done because many of these polygons will be split during the build process and we want the m_Polygons list to contain the split

geometry (the actual geometry) after the compilation process is completed. Therefore, having emptied this list, polygons will only be added back to this list when they get assigned to the nodes in the tree. These polygons may have been split several times along the way, so we will only be adding the final clipped polygon pieces back to the list. After the compile is complete, the temporary copy of the list we made will be empty (because polygons are removed as each node is constructed) and the m_Polygons list will contain the new polygon data in its split form.

```
    // Loop through all of the initial polygons
    for ( ; PolyIterator != m_Polygons.end(); ++PolyIterator )
    {
        // Retrieve the polygon
        CPolygon * pPoly = *PolyIterator;
        if ( !pPoly ) continue;

        // Store this polygon in the top polygon list
        PolyList.push_back( pPoly );

    } // Next Polygon

    // Clear the initial polygon list as this will eventually
    // become storage for whatever gets built
    m_Polygons.clear();
```

At this point we have made a copy of the entire polygon set (TreeList) that was registered with the tree and cleared the member variable list m_Polygons.

Next we kick start the build process by calling the recursive BuildTree method. It will call itself repeatedly until every node in the tree has been constructed. By the time this function returns, the entire tree will be compiled and the polygon list we just cleared will be filled with the final clipped polygon set. We pass in a pointer to the root node we just created (which will be populated by this function) and the list of polygons that are to be input into that node. For the root node, this is obviously going to be the complete set of registered polygons.

```
    // Build the tree itself
    if ( !BuildTree( m_pRootNode, PolyList ) ) return false;

    // Perform the post build steps
    return PostBuild( );
}
```

Notice also the familiar call to the CBSPNodeTree::PostBuild method after the BuildTree method returns. The PostBuild method will calculate the bounding boxes for each polygon and call the BuildRenderData method to construct the renderable versions of the data.

CBSPNodeTree::BuildTree

This is the recursive function that populates the input node. A split plane is going to be selected from the input polygon list and we will classify the remaining polygons in the list into front and back lists. Any polygons that span the plane will be split at the plane and the two child fragments added to the front and back lists appropriately. Any polygon that is found to reside on the same plane as the node (facing in the same direction) will be removed from the list and stored at the node.

The first thing this function does is select a polygon from the list that will be used to construct the node's split plane. For this task, it calls the CBSPNodeTree::SelectBestSplitter method. This function is passed the polygon list that was passed into the function and will return a pointer to a polygon that it considers the best choice for the node. We store the result in the local pointer Splitter.

```
bool CBSPNodeTree::BuildTree( CBSPNode * pNode, PolygonList PolyList )
{
    CPolygon            * CurrentPoly, * FrontSplit, * BackSplit, * Splitter;
    D3DXVECTOR3           Normal, BoundsMin, BoundsMax;
    PolygonList           FrontList, BackList;
    PolygonList::iterator PolyIterator;
    ULONG                 i;

    // Select a good splitter polygon
    Splitter = SelectBestSplitter( PolyList, 60, 3.0 );
    if ( !Splitter ) return false;
```

As you can see, the splitter selection function accepts three parameters. The first is the polygon list from which a splitter should be chosen. The second will be discussed a little later when we see the code to the SelectBestSplitter method, but essentially it allows us to control how thoroughly the passed polygon is tested for node plane candidacy. Lowering this value allows us to reduce the number of polygons in the passed list that are tested (to speed up compile times). The third parameter is a weight value that is used to bias the importance of splits when calculating the score for a polygon (described in the accompanying textbook). The higher this value, the more priority will be given to finding a polygon that creates the least number of splits; a lower priority will thus be given to finding a node that evenly divides the remaining data set into two subsets. We will look at the code to this function next, so for now just be aware that it returns a pointer to one of the polygons in the list that should be used as the node plane.

Now that we have the splitter for this node, the first thing we will do is add its pointer to the node's polygon list so that this polygon will be rendered when this node is rendered. As each node will also store a bounding box, we will also initialize two extents vectors to default values. They will be used to record the AABB of all polygons in the passed list. Keep in mind that this is the list of all polygons stored either at the node or beneath it in the tree.

```
    // Add the splitter to the node polygon list and our final list
    pNode->Polygons.push_back( Splitter );
    m_Polygons.push_back( Splitter );

    // Reset the bounding box properties
    BoundsMin = D3DXVECTOR3( FLT_MAX, FLT_MAX, FLT_MAX );
    BoundsMax = D3DXVECTOR3( -FLT_MAX, -FLT_MAX, -FLT_MAX );
```

We have the splitter polygon stored at the node but we have not yet stored the node plane itself. However, we do know what the plane normal is as it is the same as the splitter polygon's. We also know a point on that plane -- any vertex in the polygon will do. If we have a point known to be on the plane we can just dot it with the plane normal to get the distance D to the plane from the origin of the coordinate system. Since the plane normal is described in the plane equation as ABC, we now have all four plane components we need. The D3DXPlaneFromPointNormal function takes a point known to be on the plane and a plane normal and will return the <ABCD> properties in a D3DXPlane structure which we store in the node (in the CBSPNode::Plane member).

```
    // Generate the actual plane from the splitter
    D3DXPlaneFromPointNormal( &pNode->Plane,
                              (D3DXVECTOR3*)&Splitter->m_pVertex[0],
                              &Splitter->m_vecNormal );
```

We now have both the node plane and the polygon used to create it stored in the node. Our next task will be to loop through every polygon in the list and add them to either the front or back lists depending on their classification against the plane. We deliberately skip the test if the current polygon being tested is the splitter polygon as we do not want it added to any child list. Essentially, this means we have taken it out of the recursive process now that it is stored at the node. If we find any other polygons in the list that also lie on the node plane and have normals that face in the same direction, they too will be stored in the current node's polygon list. Any polygon that is spanning the node plane is split, the original polygon is deleted, and the two split fragments are added to the front and back lists.

In the first section we set up the loop to iterate through the polygon list passed into the function. If the current entry in the list is set to NULL we continue. Then, we loop through each vertex in the polygon and compare its position vector against our currently recorded minimum and maximum extents.

```
    // Classify all polygons
    for ( PolyIterator = PolyList.begin(); PolyIterator != PolyList.end(); ++PolyIterator )
    {
        // Store current poly
        CurrentPoly = *PolyIterator;
        if ( !CurrentPoly ) continue;

        // Calculate node bounding box while we're here.
        for ( i = 0; i < CurrentPoly->m_nVertexCount; ++i )
        {
            // Store info
            CVertex * pVertex = &CurrentPoly->m_pVertex[i];
            if ( pVertex->x < BoundsMin.x ) BoundsMin.x = pVertex->x;
            if ( pVertex->y < BoundsMin.y ) BoundsMin.y = pVertex->y;
            if ( pVertex->z < BoundsMin.z ) BoundsMin.z = pVertex->z;
            if ( pVertex->x > BoundsMax.x ) BoundsMax.x = pVertex->x;
            if ( pVertex->y > BoundsMax.y ) BoundsMax.y = pVertex->y;
            if ( pVertex->z > BoundsMax.z ) BoundsMax.z = pVertex->z;

        } // Next Vertex
```

At this point we have tested the vertices and adjusted the box extents vectors to contain every polygon in the list we have tested so far. Now it is time to classify the current polygon against the plane. If the current polygon is the same polygon that we used as a splitter (the same one that was returned from the SelectBestSplitter function), we skip it and continue.

```
        // Skip if this is the same as the splitter
        if ( CurrentPoly == Splitter ) continue;

        // Classify the poly against the plane
        CCollision::CLASSIFYTYPE Location =
                   CCollision::PolyClassifyPlane( CurrentPoly->m_pVertex,
                                                  CurrentPoly->m_nVertexCount,
                                                  sizeof(CVertex),
                                                  (D3DXVECTOR3&)pNode->Plane,
                                                  pNode->Plane.d );
```

At this point the local variable Location stores the classification result for the plane test. Now we enter a switch statement that handles the different cases, starting with the on-plane case.

```
        // Decide what to do
        switch ( Location )
        {
            case CCollision::CLASSIFY_ONPLANE:

                // Does the poly point in the same direction as the node?
                if ( D3DXVec3Dot( (D3DXVECTOR3*)&pNode->Plane,
                                  &CurrentPoly->m_vecNormal ) > 0 )
                {
                    // Add to node list and final list
                    pNode->Polygons.push_back( CurrentPoly );
                    m_Polygons.push_back( CurrentPoly );

                } // End if facing in the same direction
                else
                {
                    // Add straight to the back list
                    BackList.push_back( CurrentPoly );

                } // End if facing in the opposite direction
                break;
```

If the result of the classification is on-plane, this does not necessarily mean we should just store the polygon in the node. While the vertices might all share the same (node) plane, the normal of the current polygon must also be tested to see if it faces in the same direction as the node plane. We will only store polygons at the nodes when the polygon normal matches the direction of the node plane normal. Therefore, in the on-plane case we first perform a dot product between the polygon normal and the node plane normal. If the result is greater than zero then it must mean that the normal is facing in the same direction and the polygon can be added to the node. Notice that we also add its pointer to the tree's polygon list, which was emptied prior to the build process commencing. We said earlier that we only add polygons back to the main polygon list once they get assigned to a node. This is where that happens. If the polygon normal is facing in the opposing direction, we add it to the back list where it will be passed into the back child node in the next recursion. We discussed why this is the case in the accompanying textbook.

If we determine that the polygon is behind the plane or in front of the plane, it is simply added to the back or front list, respectively

```
            case CCollision::CLASSIFY_BEHIND:

                // Add straight to the back list
                BackList.push_back( CurrentPoly );
                break;

            case CCollision::CLASSIFY_INFRONT:

                // Add straight to the front list
                FrontList.push_back( CurrentPoly );
                break;
```

If the current polygon is spanning the plane then we split the polygon to the node plane using the CPolygon::Split method. This returns the two child splits in the output variables FrontSplit and BackSplit. The original polygon is then deleted from the list since it will no longer be needed and we add the front and back split fragments to the relevant lists.

```
            case CCollision::CLASSIFY_SPANNING:

                // Split the current poly against the plane and delete it
                CurrentPoly->Split( pNode->Plane, &FrontSplit, &BackSplit );
                delete CurrentPoly;

                // Add the fragments (if any survived) to the appropriate lists
                if ( BackSplit  ) BackList.push_back( BackSplit );
                if ( FrontSplit ) FrontList.push_back( FrontSplit );
                break;

        } // End Switch

    } // Next Triangle

    // Store the bounding box properties in the node
    pNode->BoundsMin = BoundsMin;
    pNode->BoundsMax = BoundsMax;
```

That ends the polygon processing loop. After the above loop exits, the front and back child lists will have been constructed and are ready to be passed into the front and back child nodes. Any polygons that were found to have matching planes with the node will not have been added to any of these lists. They will be stored at the node so that they will not be processed further down the tree. In the bottom two lines of the above code, we record the AABB extents for the polygons that made it into this node.

With the front and back lists compiled it is now time to create the child nodes, attach them to the current node and traverse into those nodes with their respective lists. In the next section of code, we begin with the front list.

```
    // Anything in front?
    if ( FrontList.size() > 0 )
    {
        // Allocate child node
        pNode->Front = new CBSPNode;
        if ( !pNode->Front ) return false;

        // Recurs into this new node
        BuildTree( pNode->Front, FrontList );

        // Clean up
        FrontList.clear();

    } // End if anything in front
```

When the recursive call to the BuildTree method returns, the entire front tree of this node will have been created. We can then clean up by clearing the front list.

Next we do the same for the back list.

```
    // Anything behind ?
```

```
    if ( BackList.size() > 0 )
    {
        // Allocate child node
        pNode->Back = new CBSPNode;
        if ( !pNode->Back ) return false;

        // Recurs into this new node
        BuildTree( pNode->Back, BackList );

        // Clean up
        BackList.clear();

    } // End if anything behind

    // Success!
    return true;
}
```

We have now seen the function that builds the entire BSP tree. Hopefully you have been pleasantly surprised by how small and familiar it is due to our coverage of the other tree types in the previous lesson. Of course, there was one function called in the above code that we have not yet covered, the SelectBestSplitter function. We will cover that next.

CBSPNodeTree::SelectBestSplitter

The CBSPNodeTree::SelectBestSplitter function is called by the recursive BuildTree function every time it needs to select a splitter polygon from the list of polygons passed into that node. In the textbook we learned that the basic algorithm to be performed is looping through every polygon and classifying its plane against all the other polygons in the list. We will record the number of polygons in the list that were found both behind and in front of the plane, which tells us the level of imbalance that will be introduced at this node should this candidate actually be chosen as the node plane. We also record the number of spanning cases that occur, which tells us how many other polygons in the current list will be split should this polygon be chosen as the splitter.

We also mentioned that the score recorded for each candidate polygon will be calculated as follows:

```
Score = (long)abs( (long)(FrontFaces - BackFaces) ) + (long)((float)Splits * SplitHeuristic);
```

FrontFaces and BackFaces hold the number of polygons that were found to be in the front and back spaces of the candidate plane, and Splits stores the number of polygons in the list that were found to be spanning that plane. SplitHeuristic is a simple scalar value that will be used to weight the importance of splits. The higher this values is, the greater the score will be with the occurrence of each split. Since it is the polygon with the lowest score at the end of the process that gets chosen as the splitter, we can see that raising this value will bias the procedure towards selecting the polygon that causes the least number of splits and lowering this value more strongly factors in the imbalance that will be caused by the candidate plane, should it be chosen. The split heuristic is a parameter to the function (we saw that we passed 3.0f in the above function) which can be tweaked during development to help you achieve a more efficient tree.

This function also takes a second parameter not discussed in the accompanying textbook, called SplitterSample. This value serves as a means for speeding up the computation of the tree by instructing the SelectBestSplitter function to only test a certain number of polygons in the list for node plane candidacy. This is very useful during debugging. For example, if this value is set to 60, then even if there are 1000 polygons in the list, only the first 60 will be tested as node plane candidates. This significantly speeds up compilation at the expense of a less efficient tree.

To understand why this feature is important you have to keep in mind that if the scene contained 50,000 polygons, when choosing the splitter at the root node we would have to test each of these polygons against all the other polygons. This would take a dreadfully long time. The children of the root (assuming a perfectly balanced tree) would then typically have somewhere in the region of 25,000 polygons, each to test against the other, and so on down the tree. We can imagine that testing every polygon that enters a node against every other polygon that enters that node is going to make the compilation process very slow on large levels. While the tree generated by sampling only a small number might be less efficient and create deeper trees, during the development cycle, we often want to run our programs as quickly as possible as we may have to test run our application many times in a single day. We really do not want to be sitting around for 15 minutes every time we run our application just because the BSP compiler is trying to choose the best possible splitter at every node. In general, we only care about getting the best tree possible when the time comes to save that data out for the release build. Therefore, the SplitterSample parameter will aid you in more rapid development. If the SplitterSample parameter is passed a value of 0, it means we wish the function to do a full test of every polygon in the passed list.

The first thing the function does is set up an iterator to step through every polygon in the passed list. Each polygon in this list will be tested as a node plane candidate. Inside this loop we fetch the pointer to the current polygon (in the pSplitter local variable) and create a plane from it.

```
CPolygon * CBSPNodeTree::SelectBestSplitter( PolygonList PolyList,
                                             unsigned long SplitterSample,
                                             float SplitHeuristic )
{
    CPolygon            * pSplitter = NULL, * pCurrentPoly = NULL, * pSelectedFace = NULL;
    long                  Score, Splits, BackFaces, FrontFaces;
    long                  BestScore = 10000000, SplitterCount = 0;
    PolygonList::iterator SrcIterator, DestIterator;
    D3DXPLANE             SplitterPlane;

    // Iterate through the face list
    for ( SrcIterator = PolyList.begin(); SrcIterator != PolyList.end(); ++SrcIterator )
    {
        // Extract the polygon we're going to test as a potential splitter
        pSplitter = *SrcIterator;
        if ( !pSplitter ) continue;

        // Create testing splitter plane
        D3DXPlaneFromPointNormal( &SplitterPlane,
                                  (D3DXVECTOR3*)&pSplitter->m_pVertex[0],
                                  &pSplitter->m_vecNormal );
```

Now that we have the node plane candidate, we will set the score variables to zero before iterating through every polygon in the list (except the one we are currently testing) and classifying them against the candidate plane.

```
        // Test against the other poly's and count the score
        Score = Splits = BackFaces = FrontFaces = 0;

        for (DestIterator=PolyList.begin(); DestIterator != PolyList.end(); ++DestIterator )
        {
            // Extract the polygon we're going to test against the splitter
            pCurrentPoly = *DestIterator;
            if ( !pCurrentPoly || pSplitter == pCurrentPoly ) continue;

            // Classify against the splitter plane
            CCollision::CLASSIFYTYPE Result =
                        CCollision::PolyClassifyPlane( pCurrentPoly->m_pVertex,
                                                       pCurrentPoly->m_nVertexCount,
                                                       sizeof(CVertex),
                                                       (D3DXVECTOR3&)SplitterPlane,
                                                       SplitterPlane.d );
            switch ( Result )
            {
                case CCollision::CLASSIFY_INFRONT:
                    FrontFaces++;
                    break;

                case CCollision::CLASSIFY_BEHIND:
                    BackFaces++;
                    break;

                case CCollision::CLASSIFY_SPANNING:
                    Splits++;
                    break;

                default:
                    break;

            } // switch

        } // Next Test Polygon
```

When the inner loop exits, we will have tested every polygon in the list against the candidate plane and recorded the number of splits that will occur and the number of polygons that will live in its front and back halfspaces. With this information, we now calculate the score for the candidate plane using the passed SplitHeuristic parameter as a weight. If the score of the polygon turns out to be lower than the lowest score we have found so far, then we record this score and the polygon. If no lower score is found, we have this information at the end of the function to return to the build process.

```
        // Tally the score (modify the splits * n )
        Score = (long)abs( (long)(FrontFaces - BackFaces) )
              + (long)((float)Splits * SplitHeuristic);

        // Is this the best score ?
        if ( Score < BestScore )
        {
            BestScore       = Score;
            pSelectedFace = pSplitter;

        } // End if better score
```

After calculating and possibly storing the score, we increment the local SplitterCount variable such that it is updated for each candidate plane we test. If the SplitterSample parameter was not set to zero (which

means we should test all polygons as node plane candidates), and if SplitterCount is greater than or equal to SplitterSample, then we have tested the number of polygons that the caller required and we break from the loop. The polygon that was recorded with the lowest score will then have its pointer returned back to the build process where it will be stored in the node and used to create the node plane.

```
        SplitterCount++;

        // Break if we reached our splitter sample limit.
        if ( SplitterSample != 0 &&
             SplitterCount >= (long)SplitterSample &&
             pSelectedFace ) break;

    } // Next Splitter Polygon

    // This will be NULL if no faces remained to be used
    return pSelectedFace;
}
```

CBSPNodeTree::PostBuild

The CBSPNodeTree::PostBuild method is called at the very bottom of the CBSPNodeTree::Build method after the tree has been fully compiled. It calls the CBSPNodeTree::CalculatePolyBounds method to calculate and store the bounding boxes for every CPolygon stored in the tree. This function is just a straight cut and paste from the CBaseTree version. The second function it calls is the CBSPNodeTree::BuildRenderData method which builds the vertex and index buffers that will be used for rendering.

```
bool CBSPNodeTree::PostBuild(  )
{
    // Calculate the polygon bounding boxes
    CalculatePolyBounds( );

    // Build the render data
    return BuildRenderData( );
}
```

When this function returns, the BSP tree will be built and all the data will be compiled into a renderable format ready for use by the application. Let us look at the CBSPNodeTree::BuildRenderData method next.

CBSPNodeTree::BuildRenderData

The BuildRenderData method is called by the PostBuild method after the tree has been constructed. Its job is to collect the vertex data stored at each node and compile it into a single vertex buffer. First it will copy all the vertex data into a D3DXMesh so that we can use its welding functionality to remove any redundant vertices in the mesh data. After the weld, the vertex data will be copied into the tree's vertex buffer and the mesh will be released. After all the vertex data has been created, this function will also allocate the dynamic index buffer that will be used by the tree during the render traversal. Finally, it will also call the recursive PopulateNodeData method which recursively walks the tree, generating the index

arrays for each polygon store at each node (in the node's CBSPRenderData structures discussed earlier). These are the indices that will be copied into the dynamic index buffer during the render traversal.

The first section of the function tests whether the device being used is capable of supporting hardware transformation and lighting. If not, the ulUsage local variable will have the D3DUSAGE_SOFTWAREPROCESSING flag set. This will be the usage parameter we use to create the vertex buffer. Notice that ulUsage is initialized at the top of the function to include the D3DUSAGE_WRITEONLY flag. This way, whether the device has hardware support or not, the buffer will be allocated optimally for write-only operations. If the device is set to NULL, we also return as it obviously means that this tree is not intended to be used for rendering and therefore no render data should be built. We also return immediately if the root node has not yet been constructed as it indicates that this function has somehow been called before the tree has been compiled.

Assuming that everything is valid up to this point, we call the CBSPNodeTree::Repair method. Although we will not cover the code to that function in this workbook (it is just a simplified version of the one used by CBaseTree), you will recall that this method removes any T-junctions that may have been introduced into the geometry by the clipping process (or perhaps unwittingly by the artist).

```
bool CBSPNodeTree::BuildRenderData(  )
{
    ULONG       nTotalTris, nTotalVertices;
    ULONG       ulUsage = D3DUSAGE_WRITEONLY;
    CVertex    *pDestVertices, *pSrcVertices;
    ULONG      *pAttributes;
    USHORT     *pIndices;
    USHORT      nCounter;
    ULONG       i, j;
    LPD3DXMESH  pMesh;
    HRESULT     hRet;

    // Should we use software vertex processing ?
    if ( !m_bHardwareTnL ) ulUsage |= D3DUSAGE_SOFTWAREPROCESSING;

    // Instructed not to build data?
    if ( !m_pD3DDevice ) return true;

    // Anything to do ?
    if ( !m_pRootNode ) return false;

    // First thing we need to do is repair any T-Junctions created during the build
    Repair( );
```

We wish to create a D3DX mesh object into which we will copy all the vertices currently stored in the tree so that we can use its weld functionality on that vertex data. Before we allocate this mesh, we need to know how big its vertex and index buffers need to be, so we will first loop through every polygon currently stored in the tree and keep a running sum of the vertices from all polygons to eventually pass into the mesh creation function. We do not have to traverse the tree to collect the polygon counts as the CBSPNodeTree::m_Polygons vector also contains a list of all the polygons pointers stored at the nodes. For each polygon, we will also add up the number of triangles it will add to the mesh since this will also be required information for the mesh creation function. Because our CPolygon structures are convex winding polygons, we can find out how many triangles we will need to create for each one simply subtracting 2 from the vertex count for that polygon. At the end of the loop shown in the next section of

code, we will have recorded the total number of vertices stored in the tree and the total number of triangles that we will need to generate indices for.

```
    // Iterate through all of the polygons in the tree and count the vertices
    // and triangles for the purposes of correctly sizing our vertex / indexbuffer.
    nTotalVertices = 0; nTotalTris = 0;
    PolygonList::iterator Iterator = m_Polygons.begin();
    for ( ; Iterator!= m_Polygons.end(); ++Iterator )
    {
        CPolygon * pPoly = *Iterator;

        // Skip if it's invalid
        if ( !pPoly || pPoly->m_nVertexCount < 3 ) continue;

        // Total up vertices and triangles
        nTotalVertices += pPoly->m_nVertexCount;
        nTotalTris     += pPoly->m_nVertexCount - 2;

    } // Next Polygon
```

With this information we can now create a D3DXMesh of the correct size. Notice that because this mesh is only being used temporarily to store data that we intend to read back from, we allocate this mesh in system memory. If the mesh is created successfully, we lock its vertex, index, and attribute buffers so that we have pointer to each for mesh population.

```
    // Create a mesh to perform a weld
    hRet = D3DXCreateMeshFVF( nTotalTris,
                              nTotalVertices,
                              D3DXMESH_SYSTEMMEM | D3DXMESH_SOFTWAREPROCESSING,
                              VERTEX_FVF,
                              m_pD3DDevice,
                              &pMesh );

    if ( FAILED( hRet ) ) return false;

    // Lock the vertex, index and attribute buffers ready for population
    if ( FAILED( pMesh->LockIndexBuffer( 0, (void**)&pIndices ) ))
       { pMesh->Release();
         return false; }

    if ( FAILED( pMesh->LockVertexBuffer( 0, (void**)&pDestVertices ) ))
       { pMesh->UnlockIndexBuffer();
         pMesh->Release();
         return false; };

    if ( FAILED( pMesh->LockAttributeBuffer( 0, &pAttributes ) ))
       { pMesh->UnlockIndexBuffer();
         pMesh->UnlockVertexBuffer();
         pMesh->Release();
         return false; }

        // Attributes should all 0 to ensure  no face remapping occurs during the weld!!!!
        ZeroMemory( pAttributes, nTotalTris * sizeof(ULONG) );
```

At the bottom of the above code, we set the attribute of every triangle in this mesh to zero so that the D3DXWeldVertices function will assume that this mesh contains a single subset. In this case we can circumvent any aggressive weld optimizations on the index data that we wish not to be altered by the process.

With our mesh now created and with pointers to its index and vertex buffers, we can loop through each polygon again and add the vertices to the vertex buffer and generate the indices for each triangle as they pertain to the vertex buffer. In the first section of the loop shown below we can see that for the current polygon being processed, we memory copy its vertex array into the destination pointed to by pDestVertices. After the vertices for each polygon have been copied, the vertex buffer pointer (pDestVertices) is incremented by the number of vertices just added such that, in the next iteration of the loop the vertices will be added immediately after the vertices of the polygon we have just added.

> **Note**: During the compilation phase, we add polygons to the tree's master polygon list at the same time we add them to the nodes. This means that the order in which the polygons are stored in the tree's master list is identical to the order in which we would visit these polygons were we to traverse the tree and process the polygons at each node. Therefore, while we are just looping through the tree's master polygon list, we are adding them to the vertex buffer in the exact order they will be visited during node traversal. This will become significant when we have to walk through the tree and calculate the index arrays of each polygon stored at the nodes, as we will see later.

```
    // Loop through each polygon
    nCounter = 0;
    for ( Iterator = m_Polygons.begin(); Iterator!= m_Polygons.end(); ++Iterator )
    {
        CPolygon * pPoly = *Iterator;

        // Skip if it's invalid
        if ( !pPoly || pPoly->m_nVertexCount < 3 ) continue;

        // Copy over the vertices
        memcpy( pDestVertices, pPoly->m_pVertex, pPoly->m_nVertexCount * sizeof(CVertex));
        pDestVertices += pPoly->m_nVertexCount;
```

Wit the vertices of the current polygon having been added to the vertex buffer it is time to generate the indices for its triangles. Recall from previous lessons that we generate the triangle indices for a convex winding polygon by stepping around its vertices in the order 0,1,2: 0,2,3: 0,3,4, etc.. Of course, since we are indexing into a global array rather than a local vertex list, the indices need to be offset to reflect prior vertices that came earlier in the global buffer. The local variable nCounter is used for this task. It will be zero for the first polygon and at the end of this loop will be incremented by the polygon's vertex count so that in the next iteration it contains the location of where the next polygon's first vertex will be placed, and so on. Here is the loop that generates the indices for the current polygon.

```
        // Build indices and attributes
        for ( j = 0; j < (unsigned)(pPoly->m_nVertexCount - 2); ++j )
        {
            // Build indices
            *pIndices++ = nCounter;
            *pIndices++ = nCounter + (USHORT)j + 1;
            *pIndices++ = nCounter + (USHORT)j + 2;

        } // Next Triangle

        // Increment counter
        nCounter += pPoly->m_nVertexCount;

    } // Next Polygon
```

At this point, the vertex buffer contains every vertex that was contained in the tree and the index buffer contains the indices of all the triangles that vertex buffer describes. In a moment, you will see that we will pass this index buffer into a recursive method that will visit each node and copy the indices for each polygon from this index buffer into the BSPRenderData structure stored at the node (in attribute order). For the moment, we have the mesh populated, so it is time to unlock the mesh buffers in preparation for the weld.

```
    // Unlock the buffers
    pMesh->UnlockIndexBuffer();
    pMesh->UnlockVertexBuffer();
    pMesh->UnlockAttributeBuffer();
```

Our mesh is ready to be welded, so we allocate a D3DXWELDEPSILONS structure and initialize all its members to have a compare tolerance of 0.001. We then pass this information along with the mesh into the D3DXWeldVertices function.

```
    // Set all epsilons to 0.001;
    D3DXWELDEPSILONS WeldEpsilons;
    float * pFloats = (float*)&WeldEpsilons;
    for ( i = 0; i < sizeof(D3DXWELDEPSILONS) / sizeof(float); i++ ) *pFloats++ = 1e-3f;

    // Weld the vertex data
    hRet = D3DXWeldVertices( pMesh,
                             D3DXWELDEPSILONS_WELDPARTIALMATCHES |
                             D3DXWELDEPSILONS_DONOTSPLIT,
                              &WeldEpsilons, NULL, NULL, NULL, NULL );

    if ( FAILED(hRet) ) { pMesh->Release(); return false; }
```

Now our vertex data will have hopefully been compacted by some amount and the indices in the index buffer will have also been updated to reflect the new positions of the post-weld vertices.

The next step is creating the vertex buffer that will be used to store the post-weld data since the mesh will be discarded shortly. In order to allocate the tree's vertex buffer we need to know how many vertices it needs to hold. We cannot use the vertex total that we calculated earlier in the function since the number of vertices may have changed due to the weld. So we will call the ID3DXMesh::GetNumVertices method to fetch the current number of vertices in the mesh and then allocate our vertex buffer.

```
    // Create the actual vertex and index buffers ready to store the data
    m_nVertexCount = pMesh->GetNumVertices();
    hRet = m_pD3DDevice->CreateVertexBuffer( m_nVertexCount * sizeof(CVertex),
                                             ulUsage,
                                             VERTEX_FVF,
                                             D3DPOOL_MANAGED,
                                             &m_pVertexBuffer, NULL );
    if ( FAILED(hRet) ) return false;
```

We will also retrieve the number of triangles in the mesh and use that to allocate the tree's dynamic index buffer. Note that we will use the D3DUSAGE_DYNAMIC flag during index buffer creation to allow later locking and filling during tree traversals without stalling the pipeline. Also notice our request for the D3DPOOL_DEFAULT resource pool. A dynamic buffer will fail to create if we ask for the

managed resource pool, which is why we will also have to manage releasing and re-allocating this buffer ourselves when the device is lost and then reset.

```
    m_nFaceCount = pMesh->GetNumFaces();
    hRet = m_pD3DDevice->CreateIndexBuffer( (m_nFaceCount * 3) * sizeof(USHORT),
                                            ulUsage | D3DUSAGE_DYNAMIC,
                                            D3DFMT_INDEX16,
                                            D3DPOOL_DEFAULT,
                                            &m_pIndexBuffer, NULL );
    if ( FAILED(hRet) ) return false;
```

We will now lock our new vertex buffer and the mesh vertex buffer and copy over the data. At this point, the vertices are in their final renderable state in the tree's vertex buffer.

```
    // Lock the vertex buffers ready for extraction
    if ( FAILED( m_pVertexBuffer->Lock( 0, 0, (void**)&pDestVertices, 0 )))
       { pMesh->Release(); return false; }

    if ( FAILED( pMesh->LockVertexBuffer( 0, (void**)&pSrcVertices )))
       { m_pVertexBuffer->Unlock(); pMesh->Release(); return false; }

    // Copy over the vertices
    memcpy( pDestVertices, pSrcVertices, pMesh->GetNumVertices() * sizeof(CVertex) );

    // Unlock Both Buffers
    pMesh->UnlockVertexBuffer( );
    m_pVertexBuffer->Unlock( );
```

With the vertex data prepared it is now time to generate the indices for each triangle. We already have the indices generated and stored in the mesh's index buffer but our render system requires that the indices of the each polygon are stored in system memory arrays at the nodes as described earlier. That way, when a node is visited during the rendering traversal, we can copy the indices into the dynamic index buffer. Because the mesh's index buffer currently contains the information we need, we will lock and pass a pointer to this index buffer's data into the recursive CBSPNodeTree::PopulateNodeData method. It will traverse the tree, starting from the root, and at each node, copy the indices from the mesh's index buffer into system memory arrays at each node. So if a node contains five polygons in its list, we will have to store five index arrays at that node. The index arrays for each polygon will be stored in a BSPRenderData array at the node and there may not be a 1:1 mapping between the polygon array and the BSPRenderData array. This is because the PopulateNodeData method will extract the indices from the mesh's index buffer in attribute order so that during the rendering traversal, we minimize the need to flush the index buffer when collecting triangles that may be spread over a range of attributes.

```
    // Lock the index buffer ready for extraction
    if ( FAILED( pMesh->LockIndexBuffer( 0, (void**)&pIndices )))
       { pMesh->Release(); return false; }

    // Populate the node's polygon data items
    bool bResult = PopulateNodeData( m_pRootNode, pIndices );

    // Unlock the buffer and release the mesh
    pMesh->UnlockIndexBuffer();
    pMesh->Release();

    // Success?
    return bResult;
```

```
}
```

Our job is now done and we can return to the caller (PostBuild function) which will essentially return program flow back to the application. At this point, the tree will be ready to render via its Draw method.

CBSPNodeTree::PopulateNodeData

This function is actually very small as it has a very simple task to perform. For each node it visits, it has to allocate an array of BSPRenderData structures at that node large enough so that there is one element for each polygon stored at that node. Because the tree's master polygon list was iterated through in order to generate the indices in the mesh's index buffer (passed into this function), this means the triangles are ordered in the passed index buffer in the exact order we would generate them if we were to walk the tree and generate the indices on a per polygon basis. Since we pass the index buffer pointer by reference, this means we can extract the indices for the current node we are visiting and then increment the pointer by the number of triangles stored at that node so that when we recurs into the next node, the indices pointer is now pointing at the index information for the polygons that are stored at that node. In short, at the start of any instance of this function, the pIndices parameter will point to the section of the index buffer that contains the triangles that were generated by the polygons stored at this node. This function relies on that fact.

We will essentially loop through each polygon stored at the node, record its vertex count, and then use this to calculate the number of triangles the polygon will generate. We will then copy over the indices from the index buffer into a BSPRenderData structure stored at the node so that at the end of this function, all the indices generated from the polygons stored at this node will have been copied from the passed index buffer into the corresponding BSPRenderData structures. As we copy over the indices for each polygon, we also total up the number of triangles we have copied into this node (nTriCount). Before we recur into the children, we can increment the passed pIndices pointer by this amount so that when this pointer is passed to the children, it points to the region of the index buffer where their triangles start, and so on.

If all we were going to do was loop through each polygon stored at the node, extract its indices from the index buffer, and copy them into the corresponding BSPRenderData structure, the function would be very simple. However, what we wish to do is copy the indices in subset order. For example, imagine that we have 50 triangles stored at the node and 20 of them use attribute 1, another twenty use attribute 2, and the other 10 use attribute 3 but that they are arranged in no particular order. We want to copy all the indices that use attribute 1 first, followed by all the indices that use attribute 2, and finally the indices that use attribute 3. So while there will still be a BSPRenderData structure in the node's array for each polygon stored at the node, there will no longer be a 1:1 mapping between the BSPRenderData array and the CPolygon list. The first section of the BSPRenderData array will store the triangles in the node that use attribute 1, followed by the triangles that use attribute 2, and so on. During the render traversal, we will then be able to collect the polygons stored at that node in subset order, minimizing the number of draw calls that have to be performed. In this example, all the polygons at the node could be rendered in three draw calls, one for each subset. If the BSPRenderData structures were not ordered by subset however, we may require many more subset changes, perhaps one for every polygon.

Therefore, our basic system will work like this. We will allocate an STL map that stores an attribute ID as the key and a Boolean as its value. Remembering that the passed index buffer pointer (pIndices) will point to the section of the index buffer where this node's regions of triangles are stored, this is what we will do:

- Fetch the number of polygons stored at the node and allocate the BSPRenderData array to store an element for each polygon.

- Loop through each BSPRenderData structure stored at the node (nBSPDataCount)

    - Loop through each polygon stored in the node and try to find an attribute ID we have not yet processed by searching the map. We skip any that we have processed already because as you will see, once we find an attribute ID we have not processed yet, we copy over the indices of all the polygons that use it.

    - At this point we have an attribute ID (AttribID) that we have not yet processed, so we add it to the map and set its value to true so that we know it has been processed and that we should not process it again.

    - We now make a temporary copy of the passed index buffer pointer (pTempIndexBuffer) so that we can repeatedly search though the section of the index buffer that pertains to this node and extract the indices for the current attribute we are processing.

    - Loop through each polygon in the node (pTest)

        - If pTest->AttribID does not equal AttribID
            - This means the current polygon we are testing does not belong to the attribute we are current trying to copy indices for.
            - Increment pTempIndexBuffer by the triangle count of pTest so that we step past the section of indices in the index buffer that were added for this polygon. We are not interested in copying over the indices of this polygon as it does not belong to the subset we are currently processing.
        - Else
            - Store AttribID in BSPRenderData structure (pBSPData[nBSPDataCount])
            - Calculate triangle count of polygon pTest and store in pBSPData[nBSPDataCount]
            - Copy over vertices from pTempIndexBuffer into pBSPData[nBSPDataCount]
            - Increment pTempIndexBuffer by ptest->TriangleCount * 3 so that it points at the indices of the next triangle in the node
            - Increase variable nTriCount which will be eventually incremented by each polygon we process. At the end of the function it stores the number of triangles generated by the node's polygons.
            - Increment outer loop variable nBSPDataCount since we have just filled out a new BSPRenderData structure.

- pIndices still points to the beginning of the section of the index buffer where this node's triangle data starts, so increment by nTriCount so that it now points at the next block of triangle data for the first child node.

- Traverse into front child with pIndices
- Traverse into back child with pIndices

Although this might seem pretty confusing at first, we are essentially just making a pass through the passed section of the index array once for each BSPRenderData structure so that we extract the triangles one subset at a time.

The first section of the code retrieves the number of polygons stored at the node currently being visited. Assuming that there is some polygon data stored at the node (in the BSP node tree case there always should be) we enter the main section of the function. The first thing we do inside this section is allocate the node's BSPRenderData array so there is one element for each polygon stored in the node. Notice at the head of the function that we instantiate an STL map called **ProcessedAttributes.** This is empty at the head of the function but will be populated by an attribute ID and a boolean set to true for each subset that is processed.

```
bool CBSPNodeTree::PopulateNodeData( CBSPNode * pNode, USHORT *& pIndices )
{
    CBSPNode::PolygonList::iterator Iterator;
    std::map<ULONG,bool>            ProcessedAttributes;
    ULONG                           nPolyCount = 0, nAttribID  = 0, nTriCount = 0;
    ULONG                           nDestPoly;
    CPolygon *                      pSrcPoly;
    USHORT   *                      pBuffer;

    // Validate parameters
    if ( !pNode ) return true;
    if ( !pIndices ) return false;

    // Get the poly count
    nPolyCount = pNode->Polygons.size();

    // Is there anything here?
    if ( nPolyCount > 0 )
    {
        // Allocate our rendering data
        pNode->PolygonData = new CBSPNode::BSPRenderData[ nPolyCount ];
        if ( !pNode->PolygonData ) return false;
```

We will now loop through each BSPRenderData structure in the array we have just allocated which forms the main outer loop of the process. Although there might seem quite a lot of code inside this loop, it is simply trying to find the next polygon we should process and thus, the next section of indices that should be copied from the passed index buffer pointer into the current BSPRenderData array.

```
        // Keep looping till we're done
        for ( nDestPoly = 0; nDestPoly < nPolyCount ; )
        {
            // First find an attribute we haven't used yet
            pSrcPoly = NULL;
```

Inside this loop we will execute a small loop through every polygon contained at the node to test their attribute IDs against the STL map. As soon as we find an attribute ID which has not yet been stored in the map we know we have encountered a polygon that belongs to a subset we have not processed yet, so we will process it next.

> **Note**: Loop variable nDestPoly contains the index of the current BSPRenderData structure we are extracting information for. You will see later how this is incremented elsewhere in the loop, so this does not mean that the number of loop iterations will be equal to the number of elements in the array. You will see how inside the loop we extract triangles on a per-subset basis and as such, if there are five polygons that use this subset, inside this loop five BSPRenderData structures will be populated and nDestPoly will be incremented by five. This loop is really to make sure that we at least loop once for every BSPRenderData structure in the worst case where every polygon in the node belongs to a different subset.

```
            for ( Iterator = pNode->Polygons.begin();
                  Iterator != pNode->Polygons.end(); ++Iterator )
            {
                // Get the polygon
                pSrcPoly = *Iterator;
                if ( !pSrcPoly ) continue;

                // Has this been used?
                if ( ProcessedAttributes.find( pSrcPoly->m_nAttribID )
                     == ProcessedAttributes.end() ) break;

            } // Next Polygon

            // Retrieve the current attribute ID and add it to the processed list
            nAttribID = pSrcPoly->m_nAttribID;
            ProcessedAttributes[ nAttribID ] = true;
```

Note the use of the STL::Map.find method which searches the map for an entry that has the passed key (attribute ID). As soon as we find an attribute that is not already in the map, we break from the loop so that we can copy the attribute ID for this polygon into the local variable nAttribID and enter its row into the map with a Boolean status of true.

We have found the next attribute we wish to extract triangle data for so we will now need to make a pass through the indices pointed at by pIndices to search for indices that belong to polygons with a matching attribute. We make a copy of the pIndices pointer in pBuffer which we can increment to step through the index array without losing our starting place.

```
            // Process all polys with this matching attribute
            pBuffer = pIndices;
            for ( Iterator = pNode->Polygons.begin();
                  Iterator != pNode->Polygons.end(); ++Iterator )
            {
                // Get the polygon
                pSrcPoly = *Iterator;
                if ( !pSrcPoly ) continue;
```

If the attribute ID for the current polygon being tested does not match the attribute ID for triangles we are looking to extract, we will just fetch the number of indices that were generated by the current polygon (VertexCount-2) * 3 and use this to increment pBuffer. This will advance us past the section of the index buffer that contains indices for the current polygon being tested.

```
                // Attribute matches?
                if ( pSrcPoly->m_nAttribID != nAttribID )
                {
                    // Just Skip over the index data for this polygon
                    pBuffer += (pSrcPoly->m_nVertexCount - 2) * 3;

                } // End if attribute does not match
```

If the attribute of the test polygon does match the attribute ID we are looking for, pBuffer will currently point to the first index in the run of triangles that were generated by this polygon. It is this section of indices we wish to extract and store in the current BSPRenderData structure we are processing.

First we get a pointer to the BSPRenderData structure in the node's array we wish to populate and then we set the attribute ID of this structure equal to the attribute ID of the polygon whose triangles we intend to store there. We also calculate the triangle count of the current polygon (VertexCount-2) and store that in the BSPRenderData structure since we will need this information when collecting the indices from this structure during the render traversal. Then we allocate the CBSPRenderData structure's index array to be large enough to store the indices of all the triangles generated by this polygon.

```
                else
                {
                    // Store pointer for easy access
                    CBSPNode::BSPRenderData * pData = &pNode->PolygonData[ nDestPoly ];

                    // Store current attribute ID, and allocate indices
                    pData->AttribID = nAttribID;
                    pData->TriCount = pSrcPoly->m_nVertexCount - 2;
                    pData->Indices  = new USHORT[ pData->TriCount * 3 ];
                    if ( !pData->Indices ) return false;
```

Next we copy over the index data from pBuffer into the BSPRenderData's Indices array before incrementing the pBuffer pointer to step past the indices we have just copied to the start of the next polygon we will process in the next iteration of the loop. We also increment the outermost loop variable nDestPoly which keeps track of the number of BSPRenderData structures we have populated so far since many may be populated in this inner polygon loop. This is the case if there are multiple polygons in the node that share the same subset. As such, if we have populated multiple BSPRenderData structures in this inner loop, the outer loop variables should be adjusted to reflect this so that we do not try to collect index data for the same structures again. We also increment the local variable nTriCount by the number of triangles generated by the current polygon we just processed. When we finally break from all the loops, this will contain the total number of triangles stored at the current node. When multiplied by three, this value describes how much we need to advance the pIndices pointer to step past the index data for this node and point at the start of the data for the front child node.

```
                    // Copy over the index data and move the buffer on
                    memcpy( pData->Indices, pBuffer, (pData->TriCount * 3) *sizeof(USHORT));
                    pBuffer += pData->TriCount * 3;

                    // We've processed this polygon data item
                    nDestPoly++;
                    nTriCount += pData->TriCount;

                } // End if attribute matches
```

```
            } // Next Polygon

        } // Keep processing

    } // End if any polys exist
```

At this point we have built all the BSPRenderData structures at this node, so we just increment the pIndices pointer to point to the start of the front child's index data. We then recurs into the front child and then the back child before returning.

```
    // Shift the indices past this block in the index buffer
    pIndices += nTriCount * 3;

    // Traverse down the front of the tree, then the back
    if ( pNode->Front ) PopulateNodeData( pNode->Front, pIndices );
    if ( pNode->Back  ) PopulateNodeData( pNode->Back, pIndices );

    // Success!
    return true;
}
```

We have now covered the entire render data build process for the BSP tree and thus are ready to examine the methods that are called to perform a back to front render traversal through the tree.

CBSPNodeTree::Draw

The CBSPNodeTree::Draw method is called by the application to set up and call the recursive process. That is, it calls the recursive CBSPNodeTree::DrawRecurse method to start the traversal at the root node. This function takes a single parameter, a reference to a CCamera class. It is this camera that will be passed through the tree and used to draw the scene into the frame buffer in back to front order. The camera class also exposes the BoundsInFrustum method that will be used by our tree to perform hierarchical frustum culling.

This function's main job is to set up the render states for the rendering of the alpha polygons and restore those states after the traversal function has returned. It also must bind the tree's index and vertex buffers to the device. Remember, this is a dynamic index buffer and it will be empty at this point.

Below we see the entire function in one block as it is very small. It enables alpha blending and sets the source and destination blend states on the device to D3DBLEND_SRCALPHA and D3DBLEND_INVSRCALPHA, respectively. As discussed in Chapter 7 of Module I these are the standard states for color blending with the frame buffer using an alpha value. We also set up the first texture stage so that its alpha pipeline modulates the texture alpha with the vertex alpha (the alpha component of the diffuse color) so that the alpha value can be extracted from either of these two sources. This is important because some of our scene's geometry may store alpha components at the vertices while others might use a texture with an alpha channel. We also disable Z writing since this is the last set of triangles that should be rendered in this scene and we know that the alpha polygons will be rendered in a perfect back to front order. Recording their per-pixel depths in the depth buffer is unnecessary work. We then bind the vertex and index buffers to the device before passing in the camera and the root node pointer to the recursive draw process.

```
void CBSPNodeTree::Draw( CCamera & Camera )
{
    // Anything to do?
    if ( !m_pD3DDevice || !m_pIndexBuffer || !m_pRootNode ) return;

    // Setup States
    m_pD3DDevice->SetFVF( VERTEX_FVF );
    m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_MODULATE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE );
    m_pD3DDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
    m_pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
    m_pD3DDevice->SetRenderstate( D3DRS_ZWRITEENABLE , FALSE);

    // Setup the index / vertex buffers
    m_pD3DDevice->SetIndices( m_pIndexBuffer );
    m_pD3DDevice->SetStreamSource( 0, m_pVertexBuffer, 0, sizeof(CVertex) );

    // Render
    DrawRecurse( m_pRootNode, Camera );

    // Restore critical states
    m_pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
    m_pD3DDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_DISABLE );
    m_pD3DDevice->SetRenderstate( D3DRS_ZWRITEENABLE , TRUE);
}
```

When the DrawRecurse method returns, all the polygons stored in the tree will have been rendered in a back to front order, so we can once again disable alpha blending, disable the alpha pipeline in texture stage 0 and re-enable depth writes. Of course, it is in the DrawRecurse method where all the real work happens, so we will look at that next.

CBSPNodeTree::DrawRecurse

Many elements of this function will look familiar to you from our coverage of our other trees' recursive functions. Notice that the parameter list is the same with the FrustumBits unsigned char initially set to zero at the root node and the bAutoVisible Boolean set to false. Please refer back to our last lesson if you do not recall the purpose of these values.

In the first section of the function you can see that if the bAutoVisible Boolean is set to true, then one of its parent nodes must have been completely contained inside the frustum and therefore, this node must be too. When this is the case, we skip the frustum test. If it is set to false however, we perform the frustum test and return from the node immediately if the node's bounding volume is found to be completely outside the frustum.

```
void CBSPNodeTree::DrawRecurse( CBSPNode * pNode,
                                CCamera & Camera,
                                UCHAR FrustumBits /* = 0x0 */,
                                bool bAutoVisible /* = false */ )
{
    ULONG                     i, nPolyCount = pNode->Polygons.size();
    CBSPNode::BSPRenderData * pRenderData;
```

```
    USHORT                  nTriCount = 0, *pIndices = NULL;

    // Perform frustum test if applicable
    if ( !bAutoVisible )
    {
        CCamera::FRUSTUM_COLLIDE FrustumResult;
        FrustumResult = Camera.BoundsInFrustum( pNode->BoundsMin,
                                                pNode->BoundsMax,
                                                NULL,
                                                &FrustumBits,
                                                &pNode->LastFrustumPlane );

        if ( FrustumResult == CCamera::FRUSTUM_OUTSIDE ) return;
        if ( FrustumResult == CCamera::FRUSTUM_INSIDE  ) bAutoVisible = true;

    } // End if
```

If we get to this point in the function, we have found that the node we are visiting is inside the frustum, so we first must classify the camera position against the node plane to determine in which halfspace it is located. If the camera position is found to be in the front half space of the node plane then it means the polygons stored at this node are facing the camera position. Therefore, as discussed in the textbook, we must first traverse into the back child to draw the sections of the tree that are furthest away from us first, then render the polygons stored at the node, and then step into the front child. If the camera position is found to be in the backspace of the plane, then we simply visit the front child first and then visit the back child. We do not have to render the polygons stored at the node in this case as it means they are facing away from us and would ultimately be back face culled in the D3D pipeline.

```
    // Classify the camera against the node plane
    CCollision::CLASSIFYTYPE Result =
                CCollision::PointClassifyPlane( Camera.GetPosition(),
                                               (D3DXVECTOR3&)pNode->Plane,
                                                pNode->Plane.d );
```

Let us now test the results of the classification of the camera position against the node plane. We will start with the case where the camera is in the front space of the plane. First we recurs into the back child.

```
    // What was the result of the classification
    if ( Result == CCollision::CLASSIFY_INFRONT || Result == CCollision::CLASSIFY_ONPLANE)
    {
        // Traverse down the back of the tree
        if ( pNode->Back ) DrawRecurse( pNode->Back, Camera, FrustumBits, bAutoVisible );
```

After the above function call returns, the entire back sub-tree of the current node will have been rendered. Now it is time to render the polygons stored at this node before we traverse into the front child.

The first thing we do is initialize a local variable called nLastAttributeID will be used to store the ID of the last triangle we added to the dynamic index buffer. As soon as we reach a point where the attribute ID of the BSPRenderData structure we are about to copy the indices from is different from this last attribute ID values, we know we have reached the subset boundary and we should render the current contents of the dynamic index buffer. So we set up a loop that will iterate for each BSPRenderData structure stored in the node and fetch a pointer to the current structure inside the loop.

```
        // Loop through the polygon data stored at this node
        ULONG nLastAttribID = INT_MAX;
        for ( i = 0; i < nPolyCount; ++i )
        {
            pRenderData = &pNode->PolygonData[i];
            if ( !pRenderData ) continue;
```

Now that we have a pointer to the current render data structure that we wish to extract the indices from into our dynamic index buffer, we must first test that its attribute ID is the same as the last render data structure we just added triangles from. If not, then it means we have to flush the current contents of the index buffer which will contain the triangles of the previous subset. So in the next section of code we examine what happens when a new subset is encountered.

Notice that the first thing we do (assuming we have already added some triangles to the index buffer) is retrieve the pointer to the attribute callback function. This is the CScene function that is called to set the texture and material on the device for the triangles we are about to render. The typedef SETATTRIBUTES is defined in CBSPNodeTree.h and is defined as a pointer to a function that takes two parameters, a void context pointer and an attribute ID. As the first parameter to the callback function we pass the context data that was registered by the scene (a pointer to the CScene object that registered the callback). The callback function will use this to access the texture and material data from its non-static array. As you can see, the inner conditional code will only be executed if we have added triangles to the index buffer, so it will never be called during the first iteration. You will see in a moment a little further down in the listing that we increment the nTriCount member every time we copy triangles into the dynamic index buffer.

```
            // Is this attribute different?
            if ( pRenderData->AttribID != nLastAttribID )
            {
                // If we have any data built so far, we must unlock the index buffer
                // and render it
                if ( nTriCount > 0 )
                {
                  // Execute the callback in order to set texture / material details etc
                  SETATTRIBUTES SetAttributes = (SETATTRIBUTES)m_AttribCallback.pFunction;

                  if ( SetAttributes )
                  SetAttributes( m_AttribCallback.pContext, nLastAttribID );

                    m_pIndexBuffer->Unlock();
                    m_pD3DDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                                        0,
                                                        0,
                                                         m_nVertexCount,
                                                         0,
                                                         nTriCount );

                    // Clear pIndices var so we know we need to lock again
                    pIndices  = NULL;
                    nTriCount = 0;

                } // End if renderable data
```

Looking at the above code it is important to realize that the inner conditional will only ever be executed if we have added triangles. In this case the dynamic index buffer will have been locked and pIndices will

point to the last place in the index buffer where we added triangles. Since the index buffer is already bound to the device (that was done in the Draw method), we can simply unlock the buffer and call the DrawIndexedPrimitive method to draw everything we have collected in the index buffer so far to the frame buffer. pIndices is then set back to NULL which we will see in a moment causes the buffer to be locked again and a new collection process started. The nTriCount member which was used to record how many triangles we have added so far will be set back to zero as the buffer has been flushed. So we can see in this section of code that the index buffer is only emptied once a subset change occurs. It is fortunate then that we have the BSPRenderData structures ordered by attribute in the node's array.

In the next section of code we can see that if pIndices is set to NULL, this is either the first time through the loop or we have just locked and flushed the index buffer. In this case, it means we wish to unlock the index buffer to be ready for another collection process. Notice that when the buffer is locked, we store the returned pointer in pIndices. It is this pointer we will use to add the indices to the index buffer.

```
                // Lock the IB?
                if ( pIndices == NULL )
                {
                    // It is very important that we test for failure here.
                    // In a lost device situation we
                    // may not have yet fully restored it before getting here.
                    if ( FAILED(m_pIndexBuffer->Lock( 0,
                                                      0,
                                                      (void**)&pIndices,
                                                       D3DLOCK_DISCARD ))) return;

                } // End if IB requires locking

            } // End if attribute has changed
```

All of the above code is only executed if the attribute ID of the current BSPRenderData structure we are processing has a different attribute ID to the last one we collected the indices from.
In the next section loop we can see that we update nLastAttribID to contain the attribute ID of the current BSPRenderData structure we are processing so that the buffer will not be flushed again until we encounter another BSPRenderData structure that has been assigned a different attribute ID. Thus we copy over the indices stored in the current BSPRenderData structure's index array into the index buffer. Notice that the location we copy the indices into is described by offsetting pIndices (which points to the beginning of the index array) by nTriCount * 3. This describes the number of indices we have added so far for the current attribute being collected.

```
            // Update last attribute ID
            nLastAttribID = pRenderData->AttribID;

            // Copy over index data and increment our renderable data count
            memcpy( &pIndices[ nTriCount * 3 ],
                     pRenderData->Indices,
                    (pRenderData->TriCount * 3) * sizeof(USHORT) );

            nTriCount += pRenderData->TriCount;

        } // Next Polygon
```

After copying the index data, we increase nTriCount by the number of triangles that were copied from the BSPRenderData structure. That ends the outer loop which is executed for all BSPRenderData structures stored in the node.

Outside the loop we unlock the index buffer and test to see if TriCount is larger than zero. If it is, then it means there are triangles in the index buffer that still need to be flushed. These will be the triangles collected for the last subset encountered in the BSPRenderData array (i.e., the triangles for the last subset stored in the BSPRenderData array) which have not have been flushed in the normal flow of the loop. This is because the buffer only gets flushed when an attribute change occurs, which for the final subset in a node, will never occur.

```
        // Unlock the indexbuffer if it was locked
        if ( pIndices ) m_pIndexBuffer->Unlock();

        // Render any remaining data
        if ( nTriCount > 0 )
        {
             // Execute the callback in order to set texture / material details etc
             SETATTRIBUTES SetAttributes = (SETATTRIBUTES)m_AttribCallback.pFunction;

             if ( SetAttributes )
             SetAttributes( m_AttribCallback.pContext, nLastAttribID );

             m_pD3DDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,
                                                 0,
                                                 0,
                                                 m_nVertexCount,
                                                 0,
                                                 nTriCount );

         }
```

Having rendered all the polygons stored at the node we then step into the front child. That concludes the case handler for the camera existing in the node's front space.

```
        // Traverse down the front of the tree
        if ( pNode->Front ) DrawRecurse( pNode->Front, Camera, FrustumBits, bAutoVisible );

    } // End if camera is infront
```

Of course, we still have to handle the case where the camera position is classified as being in the back space of the node plane. This case is easier to handle since we do not have to render the polygons stored at the node because they are facing away from us. Therefore, all we have to do is recurs into the front child and then into the back child which renders all polygons down both sides of this node.

```
    else
    {
        // Traverse down the front of the tree, then the back
        if ( pNode->Front ) DrawRecurse( pNode->Front, Camera, FrustumBits, bAutoVisible );
        if ( pNode->Back  ) DrawRecurse( pNode->Back, Camera, FrustumBits, bAutoVisible );

    } // End if camera is behind or onplane

}
```

We have now covered all the building and rendering functions of the BSP node compiler. Next we will look at the PreReset and PostReset methods of this class which as we have seen are called from the CScene namesakes when the device has been recovered from a lost state.

CBSPNodeTree::PreReset

This function is called from CScene when the device used by our BSP tree has just recovered from a lost state but has not yet been reset. This instructs our tree that any default pool resources will need to be released as they are no longer valid. As you can see, this method simply releases the index buffer interface and sets the tree's index buffer pointer to NULL.

```
void CBSPNodeTree::PreReset( )
{
    if (m_pIndexBuffer) m_pIndexBuffer->Release();
    m_pIndexBuffer = NULL;
}
```

CBSPNodeTree::PostReset

This method will be called from its CScene namesake when the device has been recovered from a lost state, has been reset, and is once again ready for use. This method will always be called after its PreReset counterpart so the index buffer of the tree will not exist at this point.

This function simply recreates the dynamic index buffer in exactly the same way that it was initially built. Once again, special attention is paid to the fact that we wish to create a dynamic index buffer which must be allocated in the default resource pool.

```
void CBSPNodeTree::PostReset( )
{
    if ( !m_pD3DDevice ) return;

    // Release previous index buffer (just in case)
    if (m_pIndexBuffer) m_pIndexBuffer->Release();
    m_pIndexBuffer = NULL;

    ULONG  ulUsage = D3DUSAGE_WRITEONLY;

    // Should we use software vertex processing ?
    if ( !m_bHardwareTnL ) ulUsage |= D3DUSAGE_SOFTWAREPROCESSING;

    // Re-Create the index buffer
    m_pD3DDevice->CreateIndexBuffer( (m_nFaceCount * 3) * sizeof(USHORT),
                                     ulUsage | D3DUSAGE_DYNAMIC,
                                     D3DFMT_INDEX16,
                                     D3DPOOL_DEFAULT,
                                     &m_pIndexBuffer, NULL );
}
```

Lab Project 16.1 Conclusion

We have now fully implemented a BSP node tree compiler and renderer. Along the way we noticed that it shares very similar qualities to the trees we developed in the previous lessons. Of course, it also has some very distinct differences, such as the polygons being stored at the nodes instead of at the terminal nodes (the leaves) of the tree. We have also had a chance to see a different type of traversal logic that can be used to render the tree in a specific order. In our example we were rendering alpha polygons and needed the back to front rendering. But by simply changing the order in which the children are traversed in the DrawRecurse method, we can easily convert it into a front to back renderer instead.

Lab Project 16.2: Solid Leaf BSP Compiler

As we move forward with the development of our graphics engine and framework, certain processes will become much more expensive for us to perform. For instance, many of the techniques employed in providing accurate scene visibility and occlusion culling schemes can involve extremely lengthy computations which cannot possibly be done in real time. In order to prevent these computations from introducing a significant delay, which could destroy the gaming experience for the player, it is important that load times remain as brief as possible. As a result, it is very often necessary to find ways to remove or optimize any processes or calculations which may need to be performed while a level is being loaded. Even in common situations, such as the compilation of a relatively simple oct-tree or kD-tree, these additional computations can easily add several seconds or more to a level's loading time. This is of course on top of the many other load-time tasks that have to be undertaken such as the reading of files, loading and potential compressing of textures, building of animation data and so on. For this reason, it will become of much greater importance that we begin to remove as much of the burden of computing such complex information from the runtime portion of our application.

Throughout the previous chapters we have introduced several new concepts and techniques which can introduce a significant delay in load time execution if we were to use a scene with a high level of geometric complexity. Constructing a spatial hierarchy from our level is by no means a trivial task as we have already discovered. In the next chapter we will be introducing the foundation of our visibility determination scheme. The operations involved in such a scheme can often take anywhere from several minutes to many *hours* to execute. To perform such lengthy operations at runtime is, as a result, unfeasible at best. One method by which we can reduce the time that it takes to load and construct our level data is to try and move as many processes and calculations as is possible from our application altogether.

As we know, most everything that we have implemented in these spatial partitioning lessons is intended to build information that allows our application to more easily manage scene data. With the exception of those processes which rely on information only available on the end user's computer hardware – such as how much data can be stored in a single vertex buffer for instance – most of the data that is constructed is based on information that does not change. As an example, during the construction of an oct-tree, kD-tree or quad-tree, the input geometry is used to define how the tree is constructed. This means that we can fully expect identical tree structures to be built on any computer system as long as they each use the same input geometry and construction parameters. Furthermore, each of these tree building classes is fundamentally assembling a series of simple data structures that are used by the application in some manner after the main game loop has commenced.

So, we know that the structure of the tree will not change irrespective of the end-user system on which it is built. As a result, it makes sense that this same information can just as easily and accurately be constructed on the developer or artist's hardware. We also know that once the game play portion of our application has begun we are simply making use of information that was built during the compilation stage. It should be possible therefore to simplify this problem by having the developer or artist save this information out to a file alongside the scene itself. This reduces our application's involvement in this process to one of simply loading and reconstructing the original data structures, rather than having to compile it on the fly. This type of operation is often referred to as a pre-process. Three such operations that exemplify such pre-process tasks – and those that will be discussed in this chapter's workbook – are

Hidden Surface Removal, *Spatial Partitioning* (of any kind) and *T-Junction Repair*. Due to the focused nature of these individual compilation tasks, it is often desirable to build a tool that is completely separate from our game application and whose sole purpose is to perform these potentially lengthy processes in advance of runtime.

With lab project 16.2, we will begin developing such a tool that will perform the brunt of our scene processing and compilation tasks. By using a relatively modular design, we will also be able to provide a level of extendibility that will be to our great benefit as we introduce additional pre-process tasks in the future.

In this section will cover the following:

- The concepts involved in building a reusable tool.
- A series of new math utility classes that will be used by this application.
- Adapting our application structures for use in a file based system.
- Building a portable pre-processing core.
- Designing a modular system for the compilation processes.
- Compilation of a polygon-aligned BSP leaf tree with solid / empty leaf information.
- Implementation of the hidden surface removal process.
- Porting the T-Junction repair system to the pre-processing tool.

Building a Reusable Tool

One of the first things that should be apparent when initially examining the source code to our new solid leaf BSP compilation tool is how different the compiler portion may appear when compared to everything we have encountered in our lab projects to date. One such difference is the implementation of several new math support classes that have been introduced in order to remove a specific reliance on the third party D3DX library provided by the DirectX™ SDK. Whilst this may seem a little superfluous to begin with, it is important to bear in mind that when we are developing a generic tool such as this we will often want to ensure that it is as portable and **re-usable** as possible.

In contrast to our game application – which is generally targeted at a specific platform / API combination – the broad applicability of the types of compilation process implemented in such tools make their re-usability highly desirable. By developing our pre-processing application in such a way that it has as few unnecessary API and platform dependencies as possible will allow this same tool to be adapted and applied to almost any project you may undertake in the future, regardless of the target platform or API. If you plan to license your toolset to third party developers, this can also be of great importance to them.

One other important benefit gained by developing our compilation toolset in this way is that it greatly increases its longevity. This is due to the fact that it will remain unaffected by any future changes made to any API on which it is dependant. Finally, by substituting the game-centric D3DX library with that of our own design we will be able to write code in a manner which best suits the specific application we are building.

The Replacement Math Classes

As a direct result of removing the dependency on the DirectX™ and associated APIs, we have introduced five new support classes which provide the math library functionality that will be required when building our compilation toolset. These include a plane class similar to D3DXPLANE, a matrix class which replaces D3DXMATRIX, two vector classes analogous to D3DXVECTOR3 and D3DXVECTOR4, and finally a new and more easily manageable axis aligned bounding box class. Each of these classes exposes member functions which provide the functionality previously accessed via calls to global D3DX functions such as D3DXMatrixMultiply in addition to other common functions that will be required.

The following list outlines each of the new math classes we will be introducing in the new compilation toolset. We will use the ‘CVector3’ class as the means by which we discuss the general design concepts used throughout our new supporting math tools. As a result we will discuss this class in greater detail than those which follow. In each case however, we will provide a summary of that class along with its intended purpose.

- **CVector3**

  This new class is designed to replace the D3DXVECTOR3 structure and associated support routines that we have been using within our run-time lab projects to date. There are many uses to

which this has been put in the past and all of the same D3DX functionality has been included here in the form of member functions. As an example of this, when performing a dot product between two D3DX vector types, we have previously made use of the 'D3DXVec3Dot' global function. With this new vector class, the dot product function has been included within the class itself via the 'CVector3::Dot' member.

In our previous implementations we have performed a dot product operation between two vectors using code similar to that shown below:

```
D3DXVECTOR3 Vector1( 1.0f, 0.0f, 0.0f );
D3DXVECTOR3 Vector2( 0.0f, 1.0f, 0.0f );

// Perform the dot product
float fDot = D3DXVec3Dot( &Vector1, &Vector2 );
```

This code should be very familiar to you at this point. As you can see we have simply built two vector classes on the stack, specifying some initial values to the class constructor. Once we have created these two vector objects, we simply pass the pointers to both of these into the 'D3DXVec3Dot' function at which point the result is returned. As mentioned previously, our new vector support class implements this same functionality using a member function rather than a global one. As a result, the code we would use in order to perform a dot product on these same two vectors now looks similar to the following:

```
CVector3 Vector1( 1.0f, 0.0f, 0.0f );
CVector3 Vector2( 0.0f, 1.0f, 0.0f );

// Perform the dot product
float fDot = Vector1.Dot( Vector2 );
```

Although the differences are relatively minor, one important thing to notice is how we have called the dot product member of 'Vector1' and simply passed 'Vector2' in as a single parameter. As before however, we are still requesting that the dot product function calculates the resulting value between the same two vectors – 'Vector1' and 'Vector2'. By using member functions in this way, we can often simplify the code that we would need to otherwise write. A good example of such a case is when we might need the ability to perform vector operations on data which is not explicitly a D3DXVECTOR3 object. Such cases have often arisen in the past when we must use *vertex* data as the input to the global vector functions. In the past, it has often been necessary to perform a complex series of casts and de-references in order to achieve this:

```
CVertex     Vertices[3];
D3DXVECTOR3 Edge1, Edge2, Normal ;

// Define Vertices
Vertex[0] = CVertex( 10.0f, 10.0f, 0.0f );
Vertex[1] = CVertex( 10.0f, 20.0f, 0.0f );
Vertex[2] = CVertex( 20.0f, 10.0f, 0.0f );

// Calculate triangle normal
D3DXVec3Normalize(&Edge1,
                  &((D3DXVECTOR3&)Vertices[1] - (D3DXVECTOR3&)Vertices[0]));
D3DXVec3Normalize(&Edge2,
                  &((D3DXVECTOR3&)Vertices[2] - (D3DXVECTOR3&)Vertices[0]));

// Cross the two edges
D3DXVec3Cross( &Normal, &Edge1, &Edge2 );

// Normalize the result
D3DXVec3Normalize( &Normal, &Normal );
```

As the complexity and length of the required operation increases, it can often become difficult to follow the indented outcome of that code. One of the primary advantages in creating a vector class in this way is that we are able to derive other classes from it. This could include our *vertex* class for instance. If we were to derive our vertex class from the new 'CVector3' class, we will gain the ability to perform vector like operations using the derived 'CVertex' object directly:

```
CVertex  Vertices[3];
CVector3 Edge1, Edge2, Normal;

// Define Vertices
Vertex[0] = CVertex( 10.0f, 10.0f, 0.0f );
Vertex[1] = CVertex( 10.0f, 20.0f, 0.0f );
Vertex[2] = CVertex( 20.0f, 10.0f, 0.0f );

// Calculate triangle normal
Edge1  = (Vertices[1] – Vertices[0]).Normalize();
Edge2  = (Vertices[2] – Vertices[0]).Normalize();
Normal = Edge1.Cross( Edge2 ).Normalize();
```

Another item of note is that the members defined within our new vector class no longer require that we pass the input parameters as pointers. Instead we have chosen to use parameter references in most of the class member functions. Recall that references are essentially the same as pointers in that they specify the memory address to an underlying data item. This provides us with most of the benefits and efficiency we would gain by using pointers but without the need to use the address-of (&) operator with every variable we use as an input parameter. This also helps us to create code which is simple and relatively easy to read and follow.

Finally, notice the method by which we have normalized the result of the cross product operation in the above code. By implementing this functionality using members within the vector class, we can often perform many operations in one go as demonstrated. This can reduce the amount of

individual steps necessary to implement such a procedure in code and can often remove the necessity that we keep temporary storage variables at our disposal for transferring the result from one step into the next. At its core, the C++ compiler is of course still creating code which allocates a temporary variable in order to achieve this. However, in this case we will arguably be able to write cleaner code, in a shorter amount of time. In addition, because we have removed the reliance on pointers and can reduce the overall amount of variables we need to keep track of, our code will often contain fewer errors resulting from incorrect casting and pointer operations or the misuse or misplacement of a temporary variable. As you might imagine, the overall complexity of the compilation tasks we will be implementing will increase dramatically as we move forward throughout this lesson and the next. This places a large degree of importance – from a coding perspective – in keeping the commonly used core libraries as clean and simple to use as possible. Hopefully, by doing so, we will eliminate many of the problems we might need to track down and debug as a result.

In the case of a game application, speed of execution is obviously paramount. To a certain degree during the development of a pre-processing tool, we do have the added advantage of being able to balance run-time execution speed against the ease with which we can develop our application. It could be argued that the compilation tasks our application must undertake may execute more slowly by not using the highly tuned D3DX mathematics library. However, this is not as crucial a concern with such one time pre-process computations as we will be developing.

Before we move on to the next new class, let us finish up by examining a selection of the functions exposed in this new implementation.

Member Functions

As we have already discussed, this new class exposes a series of member functions that provide the familiar D3DX functionality such as the dot product. The following table outlines a few of the most common functions. For a complete list of those functions defined by this class you should take a look at the ‘CVector3’ class declaration in the file named ‘CVector.h’.

Member Function	**D3DX Equivalent**
Dot	D3DXVec3Dot
Cross	D3DXVec3Cross
Length	D3DXVec3Length
SquareLength	D3DXVec3LengthSq
Normalize	D3DXVec3Normalize
TransformCoord	D3DXVec3TransformCoord
TransformNormal	D3DXVec3TransformNormal

Although this is not a complete list of the member functions implemented within the CVector3 class, this table does outline a good selection of the most commonly used D3DX equivalents. Each of these functions works in a manner similar to their D3DX counterparts with the exception

that in many cases, the result is passed back via the return value rather than by setting the contents of a variable passed into the function as a parameter.

In addition to the common vector functionality, there are several additional functions which we have developed in previous lab projects that have also been included here. As before, a sample of these functions is shown in the table below.

Member Function	Description
DistanceToPlane	There are many cases in the past where we have needed to calculate the closest distance from a plane at which a point exists. This member wraps this common functionality into a simple function that accepts the required plane as its single parameter.
DistanceToLine	In our previous implementation of the T-Junction repair process, it was necessary to determine the distance between a particular point in space and an edge / line that formed the exterior of a neighbouring polygon. This function wraps this test allowing us simply to pass in the start and end points of such a line and have it return the distance between the vector with which this member is being called and that specified line.
FuzzyCompare	Due to the nature of floating point calculations, it is never a good idea to test whether two vectors describe the same position (or direction) in space by testing for an *exact* equality between their components. We have seen that it is often necessary to use 'Epsilon' or 'Fuzzy' testing to ensure that these inaccuracies do not result in an incorrect comparison. This utility function can be used in cases where we want to test two vectors for equality, given some small tolerance value.

Again, this is not a complete list. However, this small sample should demonstrate the fact that although we have introduced several new concepts into this lab project, it is really just a rearrangement of the concepts and functions we have already developed. In doing so we make the job of developing this pre-processing tool a little more convenient. It also means that we do not have to find creative ways of integrating our rendering and *game specific* classes and utility functions into a framework with a completely different purpose.

Operator Overloads

Just as with the D3DX classes, we have also overridden several of the mathematical and logical operators common to the C++ language. We have often needed to perform mathematical operations directly on our vector structures in the past. An example of such a case is in the calculating of a direction vector by subtracting one positional vector from another. Rather than subtract each of the X, Y and Z components of these vectors individually, it has always been convenient simply to perform this operation on the vectors directly. There are other cases whereby we may want to multiply a vector by a single floating point scalar value for instance. In

this case, overloading the multiplication operator is an ideal candidate because it enables us to multiply our vector *object* by a single floating point value – something which would not otherwise be possible. The table below outlines some of the common operators that have been overloaded in this vector implementation. Again remember that this is not a full list of such operators and you should check out the class declaration for a complete list.

Operator Declaration	**Description**
`CVector3 operator+ ( CVector3& )`	This addition operator accepts a right-hand value of another vector. The implementation for this operator simply adds together each of the respective X, Y and Z components from both the left and right hand values and returns the result as a new vector object. This class also overloads the '+=' operator which functions in a similar manner. However, the right hand vector values are added directly to those of the left hand vector rather than returning a new result.
`CVector3 operator- ( CVector3& )`	Similar to the addition operator overload, this class also overloads the subtraction operator. As before we also overload the '-=' operator to allow a subtraction operation to be directly performed on an existing vector object.
`CVector3 operator* ( float& )`	This multiplication operator accepts a single floating point value as the right-hand argument. This can be used to scale a vector by multiplying each of the left-hand vector's X, Y and Z components by this single scalar value. As before, the '*=' operator for this value type is also overloaded.
`CVector3 operator/ ( float& )`	As with addition and subtraction, we also provide the complement to the multiplication operator in the form of the division operator. This accepts the same floating point argument type and, as with multiplication, simply divides the vector components by this scalar value. As with each of the previous operators, the '/=' operator has been overloaded for this same floating point argument type.

`CVector3 operator* ( CMatrix4& )`	Although we have not yet discussed our new matrix implementation, this vector class provides another multiplication operator overload that accepts a matrix as its right-hand value and returns a three dimensional output vector. As we know, there are two vector-matrix transformation functions that we have at our disposal – TransformCoord and TransformNormal. It is not possible for us to provide two operator overloads that accept the same argument types, so in the case of our vector implementation we have chosen to use the 'TransformCoord' method. This decision was based on the fact that it is often the most commonly used of the two. In the case of 'TransformNormal' however, we can still call the member function directly as we would with D3DX.
`bool operator== ( CVector3& )`	The equality operator is provided in order for us to quickly compare two vector values in a manner similar to that we would use when comparing two integer values. Unlike the aforementioned 'FuzzyCompare' function however, this operator tests for **exact** equality with no tolerance. We did this because it is a more accurate representation of what we expect from the equality operator in general. Whether you choose to use a fuzzy test in this case however, is really a matter of preference and/or convenience.
`bool operator!= ( CVector3& )`	Likewise, the inequality operator is also overloaded to allow us to test for the cases where two vector input values are *not* equal. As with the equality operator, this implementation does not use a fuzzy compare for the inequality test.

Hopefully you can see that our implementation of the 'CVector3' class is very similar to the vector concepts we are already very familiar with. We have tried to keep as close as possible to the D3DX implementation and as a result it should be a relatively simple task to adjust any existing code to use this independent vector class. In addition, by opening up the classes themselves we are able to extend them in a way that would benefit our application as we develop the modules that will rely on this functionality.

Of course, the vector class is not the only component within the D3DX mathematics library that we have come to rely heavily upon. As was mentioned at the beginning of this section, the general design of our new classes will only be discussed in relation to the 'CVector3' class. As a result we will not go into too great a detail about the layout of the remaining classes outlined

over the following pages. So without further ado, let us move on to the next of our five new support classes.

- **CVector4**

The 'CVector4' class is a replacement for 'D3DXVECTOR4' and its associated global functions. This class extends the three dimensional vector previously discussed by including the additional 'w' component as we already know. Although this particular vector class is not used directly by the application we are developing in this lesson, there are several functions available within our math library which accept a four dimensional vector as either an input or output value. An example of such a function is 'D3DXVec3Transform' ('CVector3::Transform' in our new implementation) that takes a three dimensional vector, transforms it by a 4x4 matrix and returns the resulting vector including the w component. As a result, this class has been included to allow these functions to be supported in the event that they may be required for use later in the development cycle.

Similar to its three dimensional counterpart, this class implements much of the four dimensional vector functionality exposed by D3DX within a series of member functions that work directly with the vector data. A sample of the most commonly used functions is again included below, but as before refer to the class declaration in 'CVector.h' for a complete list of those supported.

Member Function	**D3DX Equivalent**
Dot	D3DXVec4Dot
Cross	D3DXVec4Cross
Length	D3DXVec4Length
SquareLength	D3DXVec4LengthSq
Normalize	D3DXVec4Normalize
Transform	D3DXVec4Transform

Again, we are not making use of this class directly within our application at this point. However, we have also provided any appropriate operator overloads in a similar fashion to those discussed in the previous 'CVector3' coverage.

- **CMatrix4**

The 4x4 matrix is probably one of the most commonly used mathematical constructs within game development, second only to that of the three dimensional vector. It is usually quite rare for us to rely heavily upon the use of matrices during the compilation of static scene data. Although this is also true of the compilation techniques we will be implementing in this lesson, there are objects within our scene whose position and orientation are described by matrices. An example of such an object is the game 'entity' which, as we know, is essentially just a generic data storage concept providing drop in scene support for our animating objects, lights, trees etc. Although this external compilation tool will perform no rendering of the scene, this entity

information must still be processed in order for us to transfer the data into what will ultimately be exported as the compiled scene file.

Due to the complex nature of the operations that any matrix implementation is tasked with performing, it is important that the functionality offered by any such matrix class is both efficient and easy to use. Because of the fact that we are already extremely familiar with the 'D3DXMATRIX' class exposed by the D3DX math API, our matrix implementation is based heavily on this class and its associated utility functions. As with the 'CVector3' implementation of the 'D3DXVECTOR3' class, we employ a similar design in which those utility functions are provided as member functions of the matrix class itself.

Member Functions

Some of the D3DX equivalent functions that will most commonly be used are listed in the table below. In the case of our matrix class, it is recommended that you take a look at the class declaration for 'CMatrix4' included in the file named 'CMatrix.h' for a full list of member functions and their parameter layout.

Member Function(s)	**D3DX Equivalent**
Identity	D3DXMatrixIdentity
GetInverse & Invert	D3DXMatrixInverse
Transpose	D3DXMatrixTranspose
RotateAxis	D3DXMatrixRotationAxis
RotateX	D3DXMatrixRotationX
RotateY	D3DXMatrixRotationY
RotateZ	D3DXMatrixRotationZ
Rotate	D3DXMatrixRotationYawPitchRoll
Scale	D3DXMatrixScaling
Translate	D3DXMatrixTranslation
SetPerspectiveFovLH	D3DXMatrixPerspectiveFovLH
SetLookAtLH	D3DXMatrixLookAtLH

One thing that can very often be frustrating about the D3DX matrix API is the fact that almost all of the matrix functions return absolute matrices that will very often be used simply in a further concatenation with an already existing matrix. Take as an example the 'D3DXMatrixRotationAxis' function. As we know, when we examine the matrix that this function generates, we should see a set of values that describes a rotation relative to what is essentially the world axis. In most circumstances we would normally use this result in a further operation, perhaps by concatenating it with an existing matrix in order to rotate an object's world matrix by the values originally passed as parameters to the rotation function. This design can often draw out even a simple operation into a series of matrix creation and multiplication

operations where we essentially just want to rotate an existing matrix. For this reason, many of the new member functions outlined in the previous table include an additional Boolean parameter. This switch allows us to specify whether or not we would like to perform that operation on the matrix itself, rather than simply setting its values such that it *describes* the operation in question. The default in this case is to perform the operation being requested (i.e. to rotate the matrix) but this can be overridden such that the matrix is simply set in a similar fashion to the D3DX functions.

What this slightly altered design allows us to do is to request that a matrix is rotated in a single call – essentially performing the matrix multiplication on our behalf – rather than simply generating a rotation matrix that we will then have to concatenate with our target matrix manually. The code below demonstrates the code we would need to use in order to rotate an existing matrix by 90 degrees about an arbitrary axis using the D3DX library:

```
D3DXMATRIX mtxRotate;

// Generate rotation matrix
D3DXMatrixRotationAxis( &mtxRotate,
                        &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ),
                        1.5707f );

// Concatenate with existing matrix
D3DXMatrixMultiply( &mtxWorld, &mtxWorld, &mtxRotate );
```

In contrast, the following code outlines the same process as applied to our new matrix class.

```
// New method for rotating an existing matrix
mtxWorld.RotateAxis( CVector3( 0.0f, 1.0f, 0.0f ), 1.5707f );
```

This same concept as applied to each of the rotation functions is also carried forward to the 'Scale' and 'Translate' member functions. In a similar way, we can have our existing matrix physically scaled or translated, rather than simply retrieving a scaling or translation matrix for use in an additional multiplication step.

Note: While this technique can be applied to great effect, it is entirely optional. You can override this default behavior by specifying 'true' to the final 'Reset' parameter in each of these functions. This will essentially request that the matrix with which the operation is being performed is reset before calculating the result.

As with our vector implementation, there are some additional utility functions that we have integrated into our matrix class which provide quick and easy solutions to common implementation tasks.

Member Function	Description
IsIdentity	This function examines the contents of the matrix in order to determine whether or not the values describe an identity matrix. If this is found to be the case, this function returns a result of true. This can be especially useful in determining whether or not we need to use this matrix in an operation such as a recursive concatenation step or in a call to 'SetTransform' during rendering. Since the identity test itself is much less expensive than either of these operations, it can often be a good idea to test the state of a matrix using this function before it is considered.
GetIdentity	This is a **static** function that can be used to retrieve a matrix whose values have already been set to the identity state. At many points throughout the development of our applications it has often been necessary to reset – for instance – the current rendering device matrix to identity before rendering certain other objects. Instead of constructing a temporary matrix and setting its values to identity, we can simply call the 'CMatrix::GetIdentity()' function directly, passing the result into the appropriate 'SetTransform' function.
Zero	This function simply clears all internal values, setting each element in the matrix to zero.

Operator Overloads

One thing that was missing from our list of commonly used matrix member functions and their D3DX equivalents was the 'D3DXMatrixMultiply' function. Rather than include this as a class member, we have integrated this functionality by overloading the multiplication operator. Unlike the vector's 'TransformCoord' and 'TransformNormal' functions, the result expected from the matrix multiply operation is unambiguous. As a result, we simply need to provide one multiplication operator overload to add support for matrix multiplication / concatenation. The most obvious benefit of implementing the matrix multiplication operation in this way is that we can concatenate matrices very easily simply by multiplying one instance of the 'CMatrix4' class by another in precisely the same way as we would with any numeric data type as shown below:

```
mtxResult = mtxWorld * mtxTranslation;
```

Whilst there are many other operator overloads provided within the 'CMatrix4' implementation, this is by far the most important. Some of the other operators which have been overloaded include those for matrix addition and subtraction, matrix equality and inequality as well as scalar multiplication and division. For a complete list of all operator overloads provided by this class, refer to the 'CMatrix4' declaration in the project file 'CMatrix.h'.

- **CPlane3**

The plane class is one that we have focused heavily upon in recent lessons. Used throughout each of our spatial partitioning compilation and rendering procedures this is a very important class for us to implement in this processing application. In the past, we have used the D3DXPLANE structure extensively and have become familiar with it. This structure represents a plane using a 'normal' vector and a distance value. The X, Y and Z components of the plane normal in D3DXPLANE are stored in separate member variables named 'a', 'b' and 'c' in keeping with the standard plane equation. The final 'd' value describes the distance to the plane, from the origin, along the reverse of that plane's normal.

In previous lab projects we were often required to cast the plane structure to a 'D3DXVECTOR3' pointer in order to make use of some of the other D3DX global functions designed to work with vector data. In our plane implementation we are representing this information in a similar format, with the exception that the plane normal is stored in a member variable of the type 'CVector3' rather than individual floating point variables. In doing so, we allow our application to perform standard vector operations using the normal of the plane without any type of casting, extraction or manipulation of the plane data in advance. This should help simplify any operations we perform using the plane class during the development of our pre-processing toolset.

Although there are few functions available in D3DX for working with the D3DXPLANE structure, there are several plane related utility functions to which we have already been exposed.

Member Functions

During our coverage of collision detection and spatial partitioning we introduced a series of utility functions designed to classify various different primitive types against a plane. These included classification tests using a polygon, a point and even a ray. In addition, we introduced a plane technique that allowed us to detect any intersection that occurred between a ray and a plane, as well as retrieving the point at which that ray intersected. As we know, each of these tests and classification routines were used in many places throughout our previous spatial partitioning implementations. As we progress to the development of our new solid leaf BSP tree compiler, it will become apparent that this functionality is crucial to the whole compilation process. As a result of this, our plane class exposes several member utility functions that we will need at some point in the very near future:

Member Function	Description
GetPointOnPlane	As you should hopefully be aware at this point, there are two main ways in which a plane can be represented. Obviously we can represent a plane as a normal vector and a distance value, but we can also represent this as a normal and a point that lies somewhere on that plane. In order to provide a level of compatibility between these two methods, this function is provided to allow you to retrieve a point that lies on the plane described by that instance of the class.

ClassifyPoly	During the construction of the oct-tree, kD-tree, quad-tree and BSP tree we have seen how it is necessary to classify our polygons against the separating planes assigned at each node. This is used to determine the child into which certain polygons need to be passed. This function is analogous to the 'CCollision::PolyClassifyPlane' function that we have previously implemented. It returns an enumerator result informing the calling function about the position of the specified polygon in relation to the plane.
ClassifyPoint	Similar to the 'ClassifyPoly' routine, this function is intended to inform the calling function about the position of a point in space in relation to the plane itself. In previous lab projects, this function was found in the CCollision class under the name 'PointClassifyPlane'.
ClassifyRay	As with the other two functions mentioned, this is yet another primitive classification routine that we have already encountered in the 'CCollision::RayClassifyPlane' function. Once again, this function classifies the specified ray against the plane and returns a result depending on whether that ray was in-front, behind, spanning or on-plane.
GetRayIntersect	This function is the equivalent of our 'CCollision::RayIntersectPlane' function and is implemented in the same fashion. This function is called in many places, but the primary use in this lab project is during the splitting of polygon data. We will discuss where and when this splitting occurs later in this chapter.
SameFacing	During our coverage of both the node and solid leaf BSP compilation techniques in this chapter's textbook, we discussed how crucial it was that we pay special attention to the direction of the potential splitting polygons – in relation to the node planes – within the coplanar cases of our compilation techniques. This function simply accepts another plane as input and returns true if both planes point in the same direction, or false if they point in opposing directions. This should simplify several of the coplanar tests we would have to write, to a relatively simple call to this function.

We will be relying on the functionality of this class in many places throughout the development of the pre-processor application covered in this lesson. It is important that you understand how each of the member functions outlined in the above table work. If you are unsure of any of these procedures it would be a good idea to refer back to our previous coverage of these functions in addition to studying the new source code for the 'CPlane3' class found in the project files 'CPlane.h' and 'CPlane.cpp'.

- **CBounds3**

This class is a new concept and one which is not based on any structure or class that currently exists within the D3DX API. This axis aligned bounding box class wraps much of the

functionality we have come to rely on in our spatial partitioning compilation classes. In previous lab projects we have represented a bounding box using two separate 'D3DXVECTOR3' structures which describe the minimum and maximum world space extents of that bounding volume respectively. The 'CBounds3' class – the source for which can be found in the files 'CBounds.h' and 'CBounds.cpp – stores this same information in a single object. Because this bounding box information is now wrapped in a class of its own, there are several bounding box related operations that have been previously discussed which have been included in this class to provide a more integrated approach. The most important of these new member functions are described in the following table:

Member Function	**Description**
GetDimensions	This function calculates and returns the overall dimensions of the axis aligned bounding box volume on each axis. This is achieved simply by subtracting the minimum box extent values from those of the maximum box extents. As a result, this function returns a single 'CVector3' object in which the value of each component of that vector describes the dimension of the bounding box on each of its individual axes.
GetCenter	In order to perform many operations with a bounding box, we need to be able to find its center point. Because of the nature of an axis aligned bounding box, this can be determined using the following simple formula: '(Min + Max) / 2.0'. This function returns the vector that describes the position in space of the box centre point using the same process.
CalculateFromPolygon	In many cases in the past we have manually built our bounding box extent values by looping through the vertices in each individual polygon, growing the extent values if any vertex is positioned outside the box already described. Because this can be a lengthy and laborious coding exercise, this utility function wraps this process and allows us to specify a series of vertices with which it should build the bounding box values. In addition, this function also accepts a 'Reset' parameter that – when set to false – allows us to repeatedly call this function passing in a different polygon to each call and have it accumulate the results at each stage.
IntersectedByBounds	This function is our new math library's implementation of the previously discussed 'CCollision::AABBIntersectAABB' function. Put simply, this function accepts another 'CBounds3' class as an input parameter and returns 'true' if the space described by these two bounding boxes is found to overlap. Conversely, when they are not found to be intersecting, this function will return false.

PointInBounds	Our application has needed to determine whether a point falls within the space described by an axis-aligned bounding box on several occasions. In our previous lab projects this information was made available via a call to the 'CCollision::PointInAABB' procedure. This function accepts a single 'CVector3' input parameter containing the location that we would like to test against this bounding box. If the point is found to be within the bounding box extent vectors then this function will return 'true'. In all other cases, a resulting value of 'false' will be returned.
Reset	This function can be called in order to reset the bounding box member extent vectors to a set of values that can be used as the basis for the construction of a new bounding box. When initializing the axis aligned bounding box extent vectors, we will typically set each of the components of the maximum extent to a large but negative value and each of the components of the minimum extent to a large positive value. This ensures that any positional vertex or vector we consider during the calculation of the actual extent values will be taken into account. This function should be called if you plan on manually constructing the bounding box values.

This concludes the discussion of our new replacement math library. You should already be very familiar with almost all of the functionality exposed by these new classes but it was important that you understand the basic layout of each class in order to more easily follow and understand the compilation techniques we will be developing in this lesson. Before we continue on to our discussion of the code for lab project 16.2, there are one or two remaining differences that must be outlined in the tree building concepts we have been developing to date.

File Based Data Structures

One of the other key differences found when comparing this new compilation tool to the compilation techniques we have implemented to date are the means by which data structures are linked together. Each of the compiler classes we have implemented so far has been integrated directly with the game application itself. In these cases, we have been building the data structures intended for use directly by the game framework classes. In our new tool, we are building structures which are to be written out to file rather than simply maintaining that data in a memory oriented fashion.

In the past, we have often maintained links between data structures by making use of a direct pointer variable. As an example, the node classes provided by the various spatial hierarchy types we have implemented each store a pointer to other physical heap allocated child node and leaf structures. As we know, pointers are essentially variables which store the numeric starting address to the memory location in which another variable, class or structure has been allocated. When writing data to a file, we can not of course simply write the contents of a pointer variable as we would with a primitive data type such as an 'int' or 'char'. Instead we must interpret and write the data that is referenced by that pointer. While this seems simple enough, it can become quite difficult to untangle the referenced items and store them

in a manner which can be easily and efficiently reconstructed in the target application. This is especially the case when we are dealing with tree structures that are many levels deep.

When we are exporting data to file, we very often find data structures that have dependencies on one or more pieces of information – such is the case with most spatial hierarchies. In some cases there are certain pieces of information which are also shared between multiple objects. A classic example of this is with the spatial hierarchy node planes. In this case, we already know that in any individual branch of a BSP tree for instance, there should never be any other nodes that share the same plane. However there are very often many instances in which nodes are coplanar with another node *elsewhere* in the tree (e.g. further down the alternate branch of one of its parents). Due to this fact, it is entirely possible that we could select an individual node and find that it shares a plane with 10s or even 100s of other nodes in that tree.

Imagine a somewhat worse case scenario in which we found that our scene contained 10,000 unique nodes that were coplanar with exactly 9 other nodes in alternate branches of the tree. This is outside of those included in the original figure of 10,000. Even though a single plane structure occupies only 16 bytes for storage, these plane structures alone would consume 1,563kb within the file (10,000 * 10 * 16 bytes). Given that each of these structures defines exactly the same plane it should be clear that it would be much more efficient to simply store only one of these planes, and have each of our nodes simply reference that same plane data structure. If we were to do so, the planes in the above scenario would occupy only a fraction of the space at just 160kb (10,000 * 16 bytes). While this was admittedly an overly dramatic scenario, it should make the point clear that there are certain considerations that we may need to make when writing data such as this to file. Whilst it could certainly be argued that this same logic could be applied to our memory oriented data structures – and in this specific example scenario it would certainly be advisable – when working at runtime we have to balance efficiency and speed against the storage costs. However, speed of execution is rarely a concern when writing data to file and therefore, reducing storage space should certainly be one of our highest priorities.

Given the complexity of most tree structures in addition to the inherently interdependent data structure design and the potential need to reference a single piece of information from multiple items, it is often easiest for us to write data to file in contiguous blocks – whereby similar data structures are written together in one series. In essence this means that we would be exporting separate **arrays** of planes, nodes, leaves and other such data structures to file. In this case, we would simply have each item reference elements from within each of those ‘arrays’ using some form of indexing. As a byproduct of this we are also providing an easy and efficient way for our intended application to load the information back in, enabling it to load the data in a simple loop or even in some cases to read the whole block of data back into an array with a single read operation.

Although in the past we have previously constructed central arrays containing all of our leaf, polygon and detail area data structures, these existed only to provide a convenience to our spatial partitioning applications. We did this to allow our application to access all of the leaf and polygon data stored in the tree – via the base ISpatialTree interface – without having to perform a full scale traversal in order to access each of the individual pieces of information. In addition to this, our data structures referenced the components within the tree by utilizing a pointer based system. Using the node as an example – a data type which was not previously stored in a centralized array – child nodes were only ever referenced by their parent. The only means by which we could gain access to the data stored at any particular node

would be to start at the root, and traverse down the hierarchy using the child pointers stored at each subsequent node. As you might imagine, this would make it quite difficult to write the data to file for two main reasons. The first is that we have no centralized array to begin with and secondly, even if we did have such an array we would somehow have to convert any pointer which references a node within that array, into an index in order to write it to file. This would make our export procedure very complex with many procedures needed in order to search for locations within various different arrays before writing each component.

In our new file oriented tree compilation tool, the centralized arrays mentioned previously will serve as a much more integral concept. In this pre-processor application, the centralized arrays stored within the tree will become the *only* place that our data structures exist.

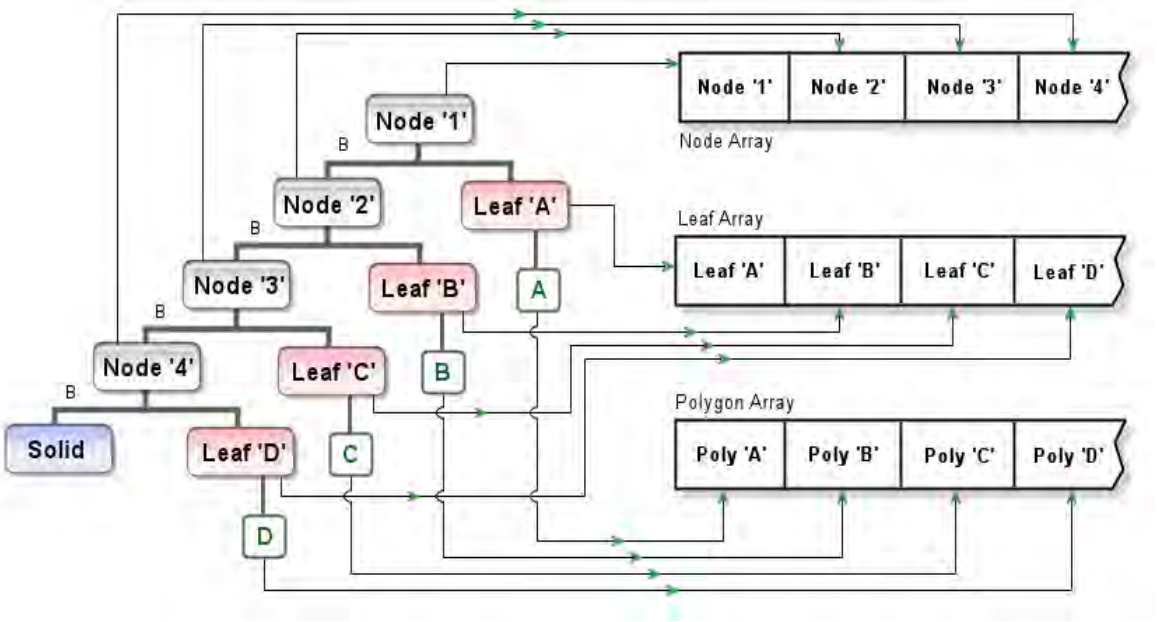


Figure 16.1

Figure 16.1 shows a number of arrays that might be used to store the tree information. We can see that each type of data structure used within the tree has its own centralized array. In addition, each instance of those data structures has a unique position within that array. Recall that we need to move away from the use of pointers in our data structures if we want to easily write the data to file. The upshot of reorganizing our compilation code in this way is that we can begin to use indices that direct us to the correct element in each of these arrays rather than using pointers (physical memory addresses):

```
class CNode
{
public:

    // Public Variables for this Class
    int Front;
    int Back;

    ...
};
```

This pseudo-class demonstrates how we might employ the use of indices within one of the most common tree data structure types. In previous implementations, the child member variables stored within the node were direct memory pointers. In this case, these pointers have been replaced with simple numeric variables of the type 'int'. If we ensure that any new data items are placed into a centralized array at the same time as we allocate that item it is guaranteed that we will have access to the correct array index which can be stored in the data structure itself. The following pseudo-code is an example of how this might be achieved.

```
// Allocate new node
pNewNode = new CNode;

// Setup new node data
...

// Attach to the front using its position as
// it will exist in the array (the next item)
pCurrentNode->Front = m_Nodes.size();

// Add the node to the centralized array
m_Nodes.push_back( pNewNode );
```

In this example we are creating a new front node in preparation to recurse into a new level of the tree construction process. In this code, the first thing we do is to allocate a new node. Imagine at this point that we had already stored 10 nodes in an STL vector called 'm_Nodes'. Recall that array indices are zero-relative which means that the first element in the array can be accessed using index '0', and likewise the 11th element by using index '10'. The very next line of code retrieves the *current* size of the node vector. Because we have not yet added the new node to the array – which as we know contains ten nodes already – the return value from the STL vector 'size' call will be 10. The final line in this example adds the new front node to the end of the vector as the 11th element in the array. Given the zero-relative nature of array indexing it should be clear that the index of '10' stored in the previous line is correct.

There are other types in addition to the node which we will of course have to adapt to this file friendly tree organization approach. These include both the leaves attached to the nodes and the polygons stored within those leaves. With the leaves in particular, we previously created unique child 'leaf-nodes' within our memory based tree structure whose primary purpose was to store a pointer to the leaf that existed there. During the compilation process, these leaf nodes were attached to the relevant child pointer of the current node enabling us to greatly simplify our traversal and construction logic. When written to file, these additional leaf nodes serve no informational purpose and as a result the BSP tree building class we will construct in this lesson does not include these additional nodes. Instead we will be taking advantage of the transition from pointers to indices in order to link the leaves directly to the parent node structures.

The easiest way to achieve this is to use a *signed* data type for the child index member variables stored within the node structure. In the previous example of how we might declare our index based node structure we used the signed type 'int' for the 'Front' and 'Back' member variables. As we know, a node needs to be able to reference two types of children. These are either another child node, or a leaf structure. A 32 bit *unsigned* integer variable is capable of storing values up to and including 4,294,967,295. Since it is not possible for us to ever create anywhere near this number of nodes, it is

safe to assume that we can use a *signed* data type which is capable of storing values between +/- 2,147,483,647. Using standard indexing procedure we know that we can use the positive index stored within a child member variable of a node to reference into our node array. However, since we are using a signed data type we can also use the negative range of values to signal to our traversal procedures – as well as our level import process – that once we convert it back to an absolute value, this member variable is an index into the *leaf* array instead.

One small caveat that you may have already spotted in this procedure is that when indexing into an array the first element is accessed with an index of 0. However, if the node's child index stores a zero value then there would be no way to know whether it is pointing to the first element in the leaf array or that same element in the node array. As a result of this we must make a slight adjustment to the signed / unsigned index ranges used such that nodes are indexed with values of 0 and above, and leaves are indexed using values from -1 and below. Take a look at the code below in which we show an example of how we might generate and store the index for a leaf in its parent node, using this signed / unsigned concept.

```
    // Allocate new leaf
    pNewLeaf = new CLeaf;

    // Setup new leaf data
    ...

    // Attach to the front using its position as
    // it will exist in the array (the next item)
    // but altered such that we use our negative range
    pCurrentNode->Front = -(m_Leaves.size() + 1);

    // Add the leaf to the centralized array
    m_Leaves.push_back( pNewLeaf );
```

As before, we first create our leaf object and set up its values as necessary. Again, on the next line we want to link this leaf to the current node's front by storing an index. In this case, imagine that we have not yet stored any leaves in the 'm_Leaves' vector. Because of the fact that the vector's 'size' function will return a value of 0 at this point, when we add 1 to this value and invert it's sign we would be storing a value of '-1' in the current node's front child member.

So, given that we are now storing indices in our node structure to both its child nodes in addition to directly referencing leaves, how might we go about turning these values back into physical array indices? The following pseudo-code example demonstrates just this.

```
    // Is there a leaf or a node in front of the current node?
    if ( pCurrentNode->Front < 0 )
    {
        // Get the leaf index
        int LeafIndex = abs( pCurrentNode->Front + 1 );

        // Retrieve the leaf
```

```
        CLeaf * pLeaf = m_Leaves[ LeafIndex ];

        ...

    } // End if leaf
    else
    {
        // Get the node index
        int NodeIndex = pCurrentNode->Front;

        // Retrieve the node
        CNode * pNode = m_Nodes[ NodeIndex ];

        ...

    } // End if node
```

As you can see, we can now detect whether a node is referencing a child leaf or another child node by testing the sign of the index stored in the relevant child member variable. If this value is found to be less than zero, then the variable is referencing an element into the leaf array. Conversely if it is found to be greater than or *equal to* zero, then we know that this must be referencing a child node. Due to the ranges that were chosen to describe each type of reference, when this child is found to be a node, we need do nothing with the index stored there. In this case, we can simply use the index to retrive the node from our centralized array directly. If it is a leaf however, we must undo the adjustment to its value that we made when storing the index, as well as flipping its sign such that we have an absolute index value. This is done in the following line of code.

```
int LeafIndex = abs( pCurrentNode->Front + 1 );
```

In this line we take the current value stored in the node's child member variable and add 1 to it before we retrieve the absolute of that resulting value. It may initially seem strange that we are again adding 1 to this value as we did when we were calculating the index for storage. Remember however, that the entire result of the original addition was negated just before we stored it. Before we move on, let us just test this code by using some real values. If you are not sure at this point exactly how this procedure works, then following through the example using real values should be helpful. The first example below demonstrates the initial calculation of the negative value that will be stored in the node when referencing a leaf. For the purposes of this example, assume that the leaf array already contains 10 other leaves.

```
Front = -(m_Leaves.size() + 1)

=
Front = -( 10 + 1 )

=
Front = -( 11 )
```

As you can hopefully see, even though our node will need to index into the leaf array using an index value of 10, we actually store a value of -11 in the node. Obviously this means that if we needed to index into the first element in the leaf array (index 0) that this same logic would generate a value of -1 as

we discussed previously. Let us finish off this example by retrieving the correct index based on the value that we have just calculated.

```
LeafIndex = abs( Front + 1 )

=
LeafIndex = abs( -11 + 1 )

=
LeafIndex = abs( -10 )

=
LeafIndex = 10
```

Following this example through demonstrates that the returned index value references into the leaf array using an index value of 10 as we had expected.

Although this does add a level of complexity to both our compilation and import processes, we are able to eliminate a number of nodes equal to that of the number of leaves from the resulting file. Given that the storage of an average node data structure might be in the area of 36 bytes per element or more, this would save us over 100kb in our final export file for a level containing 3000 leaves. Since these nodes do not give us any particular benefits within the file itself, this is an important storage optimization.

With these alterations made to the memory based structure we have previously used in our run-time projects, we should be able to quickly and efficiently export the tree data to file. Although the exact implementation details will ultimately be discussed as we move through the code and structures of our new application, this section should have made you aware of the implications of and reasons why we have moved to an array and index solution in our new pre-processing tool. As we move forward with this lesson, we will discover that our compiler tools make heavy use of this concept throughout and – as mentioned – will be heavily biased toward the way in which it will finally be exported. Before we finish up this discussion thefore, let us take a look at the final layout of the data as it will be written to file when saving the leaf tree information.

Table 1: Plane Data Layout

Name	Description	Type	Size (in bytes)
PlaneCount	The number of planes that have been written to file.	unsigned long	4
Repeated for each plane as described by the previously written 'PlaneCount'			
PlaneNormal	The normal of the plane being described.	CVector3	12
PlaneDistance	The plane distance / 'd' component of the plane.	float	4
End of 'PlaneCount' repeat.			

Table 2: Node Data Layout

Name	Description	Type	Size (in bytes)
NodeCount	The number of nodes that have been written to file.	unsigned long	4
Repeated for each node as described by the previously written 'NodeCount'			
PlaneIndex	Index into the previously written plane data / array that describes the plane used to construct this node. Note: Multiple nodes may reference the same plane.	long	4
BoundsMin	The minimum bounding box extents of this node. This describes the extent of all of the polygons contained below it in the hierarchy.	CVector3	12
BoundsMax	The maximum bounding box extents of this node. This describes the extent of all of the polygons contained below it in the hierarchy.	CVector3	12
Front	The front child index for this node. This can be either a positive or negative number as described previously.	long	4
Back	The back child index for this node. This can be either a positive or negative number as described previously.	long	4
End of 'NodeCount' repeat.			

Table 3: Leaf Data Layout

Name	Description	Type	Size (in bytes)
LeafCount	The number of leaves that have been written to file.	unsigned long	4
Repeated for each leaf as described by the previously written 'LeafCount'			
BoundsMin	The minimum bounding box extents of this leaf. This describes the extent of all of the polygons stored within this leaf alone.	CVector3	12
BoundsMax	The maximum bounding box extents of this leaf. This describes the extent of all of the polygons stored within this leaf alone.	CVector3	12
PVSIndex	This is used in the next lesson, for now it should be considered as a 'reserved' data area and should be written with a value of '0'.	unsigned long	4
PolygonCount	The number of polygons that are stored within / referenced by this leaf.	unsigned long	4
PortalCount	This is used in the next lesson, for now it should be considered as a 'reserved' data area and should be written with a value of '0'.	unsigned long	4
Reserved	This value is literally reserved for later use / expansion. This should be written with a value of '0'.	unsigned long	4
Repeated for each polygon as described by the previously written 'PolygonCount'			
PolygonIndex	Index into the main polygon array that was written into the **first** mesh in the IWF file.	unsigned long	4
End of 'PolygonCount' repeat.			
End of 'LeafCount' repeat.			

Navigating the Compiler Project Source Code

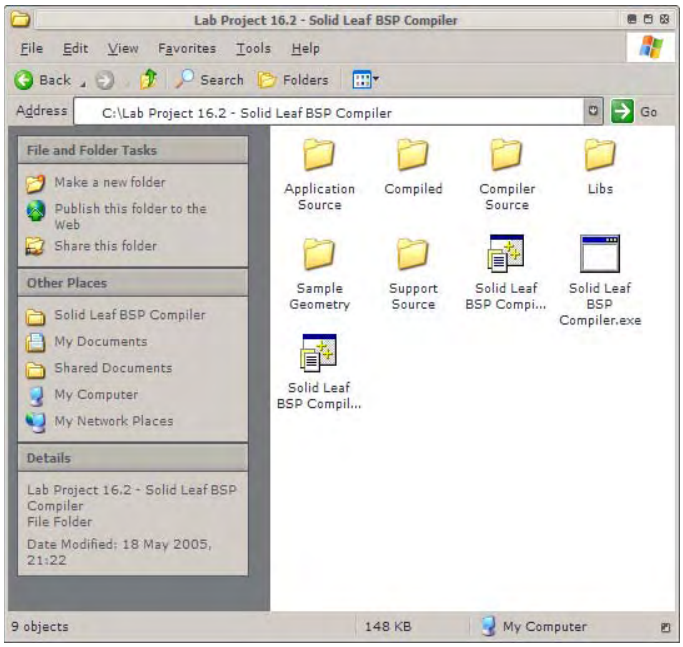


Figure 16.2

Because we are building a brand new application in lab project 16.2 and one that is not based on the previous demo framework we have used in the past, there are some changes to the layout of the project directory structure. As was mentioned previously, the pre-processing application that we are building in this lesson is designed to be as modular as possible. This will allow us to add additional features and compilation procedures to the application in the future without having to rewrite large portions of code. In addition to this, we have tried to design a portable system whereby the core compilation portions of the application is completely separated from whatever front end interface might be desired. In certain cases there are situations in which some form of feedback is required by the application. One example of this is how the application displays the current progress and outcome of each compilation task. As we have discussed, some compiler computations can take a long period of time to complete. If our application simply displayed a blank screen until the scene was completely processed, the user might reasonably believe that the application had stopped responding. Due to the fact that we want to keep the compiler completely separated from the application front end, and that only the compiler is really in a position to inform the user about how each task is progressing, this presents us with a problem. We can overcome this situation however, by defining a base interface – which we have named 'ILogger' – that both the front end and the compiler are aware of. The application is then responsible for deriving a class from this interface and its defined functions, which displays the logging information in any way it requires. Because the compiler uses the abstract 'ILogger' interface to output any progress information, we preserve the independence of the compiler and our ability to insert it into any application.

At this point, rather than become distracted by a comprehensive and potentially complex full blown GUI front end, we have opted for a relatively simple console based framework in which we will insert the compiler code. We have also developed a derived logging class which simply outputs the progress information directly to the console window itself.

It is clear that we have two separate components at this stage. We have the application framework and its associated front end, in addition to the compiler logic and compilation process source code. We have also introduced several new support classes however – such as our new vector, matrix, AABB and plane classes – that bring several new header and source files with them. In an effort to ensure that our pre-processing tool source code can be more easily managed, each of the source components that make up our application has been separated into various directories.

Although it should be relatively easy to find your way around the source code folder, the following list outlines the contents of each of the directories in the new project structure in addition to the intention behind them should you need to add additional source files of your own.

Application Source

This directory contains the main source files and headers related only to the application framework and interface front end that will create and call the compiler classes and functionality. The intention behind this directory is that it should contain the source files and header files that define the application entry point, any sort of display procedures, graphical user interface classes and anything which is not part of the core compiler itself. In this lab project, there are two main source file and header combinations located in this directory:

Main.cpp & Main.h
These source files define the application entry point (the *WinMain* function) which is ultimately responsible for setting up and calling the compiler class functionality. These files are relatively simplistic but are responsible for handling how the compiler is set up as well as the main logic of the application execution.

LogOutput.cpp & LogOutput.h
These source files define the derived 'ILogger' class named 'CLogOutput'. This class's purpose is to output the progress information to the application's console window, as reported by each compilation process. This class implements the pure virtual functions declared by the 'ILogger' interface.

Compiler Source

This directory is the home of our compiler application's core processing source code. Any source code files relating to the individual compilation tasks such as the BSP compilation, hidden surface removal and T-Junction repair procedures included in this lab project can be found here. This is also the directory into which we will place any additional modules that will be developed in later lessons. In addition, all source files relating to the import and export of the scene and compiled data can be found here.

Compiler.cpp & Compiler.h
These files define the class 'CCompiler'. This class exposes the primary compiler API that our application front end will use in order to perform the core tasks of our pre-processing tool. For instance, this class includes functions which allows the application choose which compiler tasks to perform, to set any applicable compiler options and to instruct the compiler component to load and save a particular file. The application will never typically work directly with the individual compiler process classes (outlined shortly), but will instead work through the interface defined by this class.

CompilerTypes.cpp & CompilerTypes.h
There are many classes defined by these files which include items such as 'CMesh', 'CPolygon' and 'CVertex'. In each case, the classes and structures defined here are dependencies of the compiler code. In most cases, these classes are used to house any data that has been loaded from the scene file specified

by the application. Also defined here are the various options structures and miscellaneous definitions that are used by the application to setup the various compiler processes.

CFileIWF.cpp & CFileIWF.h
The 'CFileIWF' class is defined in these files. This class is responsible for loading and saving our scene data both before and after compilation. Although we have previously been using the 'CFileIWF' class as provided by the static 'libIWF' import library, in this project we also need to include the file export logic that is executed once the compilation procedures have completed and the application requests that the compiled file is saved. In addition we are using several custom classes – such as the custom mesh, polygon and vertex classes discussed later. Rather than complicate matters by using intermediate data structures for importing and exporting the tree data, we will implement the 'CFileIWF' class directly here in order to extend it and allow us to use our custom classes directly.

CBSPTree.cpp & CBSPTree.h
The classes defined in these files are the very foundation of everything we will be developing in both this lesson and the next. These classes include our main solid BSP leaf tree compilation class (CBSPTree) in addition to the leaf (CBSPLeaf), node (CBSPNode) and other associated classes. The 'CBSPTree' class also serves as the central hub for storing most of the information which we will be building throughout the various compilation tasks.

ProcessHSR.cpp & ProcessHSR.h
Because our compiler tool is designed to be modular, each of the individual compilation processes is separated out into individual source files and classes. These files define the 'CProcessHSR' class which is home to the hidden surface removal processing logic. This module is created and called by the aforementioned 'CCompiler' class during the overall build process.

ProcessTJR.cpp & ProcessTJR.h
As with the HSR process, the T-Junction repair step is segregated into its own modular class. This class as defined within these files is named 'CProcessTJR'. Again, this class is instantiated and executed by the 'CCompiler' class during scene compilation. Although we will implement additional processing modules in the future, this process is generally the one that will be executed as the very final step before saving.

Support Source

The support source directory contains all of the miscellaneous supporting classes used by the compiler in addition to any header files which may be common to both the application front end and the compiler classes. Recall that we previously discussed the inclusion of a new supporting math library which included the classes 'CVector3', 'CVector4', 'CMatrix4', 'CPlane3' and 'CBounds3'. Although not listed below, this directory also contains the source code implementations for each of these new classes.

Common.h
This is the common header file which links the front end application with our compiler core. This file includes several macros and other useful definitions that are used by both components of our application. It also includes certain other headers which will need to be in place for the compiler to

integrate into the application without issue. This header is also one in which the 'ILogger' interface is declared, from which the application should derive its progress display class.

AppError.h
Last but not least is the header that contains all of the potential error definitions used by the compiler and available for use by the application. Modeled after D3D's HRESULT error codes, these types of error descriptions should be familiar to you. Although we are using a 'result code' system similar to that of D3D in this application, for the most part these errors are not used as return codes. Instead the compiler makes extensive use of exception handling. As a result, whenever we need to handle errors which occur within the compile process, we include a catch block using an argument type of 'HRESULT' which is also defined here.

Now that we are familiar with the new project layout and design, it should be a little easier to navigate to the various portions of the source code with greater ease as we progress onto the implementation of our new tool. As such, let us finally move on to the next section in which we will begin to focus on the actual implementation of some of the classes we have discussed.

Scene Data Classes – CompilerTypes.cpp

Before we can realistically move on to the implementation of our new compiler application, there are several scene related classes that need to be discussed first. Throughout the development of the many lab projects we have built in the previous lessons of this series, we have created many classes whose purpose is to create, store and render the scene data. These included classes such as CMesh, CPolygon and CVertex. In each of these cases, a lot of thought had to go into developing a system which could be easily integrated into our application and was able to construct data in a hardware friendly fashion. We also developed several other structures and classes that stored and processed any secondary data such as texturing and material information. Obviously, these previous applications were heavily biased toward the rendering of a scene. As a result, a lot of work had to be done in order to prepare the information contained within the 'IWF' scene file that would ultimately allow our application to display the game world correctly.

In this application the primary focus is on importing and storing the data for use by the various compilation procedures. As a result we are able to simplify most of the scene data concepts we have previously employed, into one that simply provides a logical place for storage and that grants our tool easy access to that information. There is an additional step in this new project that will make use of the scene data however. In this tool we must also *save* the scene to file. This is something that we have not had to consider in those previous rendering applications. Because of this additional process, there are several pieces of information that must be imported and maintained that have no other purpose than to be exported back out to the resulting file. Some examples of this type of data are: surface textures and materials, any shader references that may be contained within the file and also scene entity information which we have come to rely so heavily upon in the various rendering projects.

The tools that we have traditionally used to load all of our scene information have always been those exposed by our static IWF import library. Recall that the main functionality needed for the importation of the scene data from the IWF file was made available to us through the 'CFileIWF' class and its support structures. We have previously touched upon the fact that we will be implementing our own

custom version of that class in this project. We need to do this because there are several custom scene data storage classes which are being utilized within the tool. In addition, our application is of course now required to add export functionality to this file handling class. Shortly we will be discussing the classes and structures we will implement for the data types used directly by our new compiler application. However, rather than re-implement each of the data classes that our application will **not** make direct use of – e.g. materials, entities, textures and shader information – we will rely once again on the data classes exposed by the import library. Since we have previously covered most of the support classes provided by the libIWF library, let us just quickly recap on each of the data classes that our pre-processor tool will employ.

- **iwfMaterial**

```
class iwfMaterial
{
public:
    // Constructors & Destructors for This Class.
     iwfMaterial( );
    ~iwfMaterial( );

    // Public Variables for This Class
    char           *Name;           // Material Name (if applicable)
    COLOUR_VALUE    Diffuse;        // Diffuse reflection component
    COLOUR_VALUE    Ambient;        // Ambient reflection component
    COLOUR_VALUE    Emissive;       // Emissive reflection component
    COLOUR_VALUE    Specular;       // Specular reflection component
    float           Power;          // Specular reflection power ratio
};
```

This is a class that we have already encountered in previous demo lab projects. The data members declared within this class closely represent those defined by the Direct3D 'D3DXMATERIAL9' structure and is used almost exclusively in the calculation of scene lighting information. The member variables declared by this class include the ambient, diffuse, specular and emissive surface reflection properties with which you should be familiar. Each of these material properties are common throughout many of the available vertex and pixel lighting techniques and should therefore also be portable to other immediate mode rendering APIs. As with each of these data storage classes, the declaration for this class can be found in the 'iwfObjects.h' header file in the project's 'Libs' subdirectory.

- **iwfEntity**

```
class iwfEntity : public IIWFObject
{
public:
    // Constructors & Destructors for This Class.
             iwfEntity( ULONG Size );
             iwfEntity( );
    virtual ~iwfEntity( );

    // Public Variables for This Class
    UCHAR           AuthorIDLength; // Length of the author ID code
    UCHAR           AuthorID[255];  // Actual author ID data.
```

```
    USHORT          EntityTypeID;   // The Type Identifier for the entity.
    char           *Name;           // Entity Name
    MATRIX4         ObjectMatrix;   // The matrix assigned to the entity.
    ULONG           DataSize;       // The size of the entities data area
    UCHAR          *DataArea;       // The entities data area.

    // Public Member Functions Omitted
};
```

As with the aforementioned 'iwfMaterial', we have seen and used the 'iwfEntity' class many times in previous lab projects. As we know, the entity type is a generic object concept that can be used to store many different types of information. We have used this class for objects such as lights, fog, trees, references, external meshes / characters, terrains and even a skybox. In this application we will not interact directly with the imported entity objects. This class will simply be used to store the imported entity data until such time as the scene has been processed, and the application chooses to export the scene to another file.

- **iwfTexture**

```
class iwfTexture
{
public:
    // Constructors & Destructors for This Class.
             iwfTexture( ULONG Size );
             iwfTexture( );
    virtual ~iwfTexture( );

    // Public Variables for This Class
    UCHAR   TextureSource;  // Information about where the texture is
    char   *Name;           // The texture name.
    UCHAR   TextureFormat;  // The format of the internal texture if used
    USHORT  TextureSize;    // Size of the TextureData array
    UCHAR  *TextureData;    // The data being referenced

    // Public Member Functions Omitted
};
```

Although we have not used the 'iwfTexture' class itself in the past, this class is just a more object oriented version of the 'TEXTURE_REF' structure that we have used in almost every lab project we have developed from chapter five onwards. This class is designed to store various pieces of texturing information, but the most common use is one in which each instance stores the filename of a scene texture to be applied to the appropriate pieces of level geometry that reference it. As with the material and entity information, this application will not interact directly with objects of this type. This data will simply be imported along with the scene data, and exported again when the processed scene is saved.

- **iwfShader**

```
class iwfShader
{
public:
```

```
    // Constructors & Destructors for This Class.
             iwfShader( ) {};
    virtual ~iwfShader( ) {};

    // Public Variables for This Class
    ULONG           Components;     // Which components are included
    iwfScriptRef    VertexShader;   // The vertex shader if included
    iwfScriptRef    PixelShader;    // The pixel shader if included
};
```

We have not previously used the shader classes or structures exposed by the libIWF import library, but they are maintained within this application in order to allow for future enhancement. The 'iwfShader' class itself does not store any specific pieces of shader information. Instead it declares two members named 'VertexShader' and 'PixelShader'. Each of these members is of the type 'iwfScriptRef' outlined below. Once again this information is only imported for the purposes of being transferred to the export file once the scene has been compiled.

```
class iwfScriptRef
{
public:
    // Constructors & Destructors for This Class.
             iwfScriptRef( ULONG Size );
             iwfScriptRef( );
    virtual ~iwfScriptRef( );

    // Public Variables for This Class
    UCHAR   ScriptSource;   // Information about where the script is
    char   *Name;           // The script name
    USHORT  ScriptSize;     // Size of the ScriptData array
    UCHAR  *ScriptData;     // The data being referenced

    // Public Member Functions Omitted
};
```

This class is intended to reference script files or script data in much the same way as the 'iwfTexture' class references texture files. While this structure is used independently in other locations within the IWF specification, the script reference concept is used only for storing shader file references in this lab project.

If you would like further information about the various structures, flags and concepts surrounding each of these IWF data classes it would be a good idea to refer to the IWF SDK documentation available from the classroom download area.

The CVertex Class

We should already be extremely comfortable with the concept of a vertex class because we have used them extensively since the very first chapter. In each of the lab projects to date however, our vertex class has been used primarily for storing data in a format best suited for the purposes of rendering, or building vertex buffer data. In this application we are focused more on the storage and ease of use aspects due to the fact that our pre-processing tool will have no rendering component. As a result, there are a few

changes which have been made to the vertex class concept that should make the task of developing this new application a little easier.

```
class CVertex : public CVector3
{
public:
    // Public Variables For This Class
    // (X/Y/Z components inherited)
    CVector3    Normal;    // Vertex Normal
    float       tu;        // Vertex U texture coordinate
    float       tv;        // Vertex V texture coordinate
```

The first thing to notice about the declaration of our new 'CVertex' class is that it is derived from one of the new math support classes discussed earlier in this chapter. As you can see, we derived this class from 'CVector3' which if you recall contains most of the vector functionality previously offered by the D3DX math component. By deriving the vertex class from 'CVector3' in this way, our compiler component is able to interact with either a vertex or vector object in exactly the same way. In addition, we are able to pass any *vertex* object into a function that is expecting a vector typed parameter. As an example, using this class hierarchy we might perform a dot or cross product on the vertex data directly. We could also classify a vertex against a plane without any of the confusing casting operations that we would previously have included with our standalone vertex structure. In essence, our vertex class is simply extending the functionality of the vector, allowing us to integrate the two concepts more easily.

In this portion of the class declaration it is important to notice that the X, Y and Z components of the vertex have not been included as data members. This is due to these positional components having already been declared by the new three dimensional vector class from which it is derived. By doing so, the vertex class inherits these three public member variables from the vector and as a result they should not be declared again. This is the very thing that allows our vector class functionality to work directly with the positional component of our vertices.

Although there is no direct implementation of the 'CVertex' within any source module (.cpp file) – outside of those member functions inherited from the vector class – there are several constructors defined directly within the 'CompilerTypes.h' header file:

```
    // Constructors & Destructors
    CVertex( ) { x = y = z = tu = tv = 0.0f; }
```

The first of these is the default constructor for this class. This function accepts no input parameters and simply resets each of the vertex component values to zero. Notice how in this function we are setting the inherited vector component values to zero in addition to those declared by the vertex class itself (the texture coordinate values tu & tv). One thing that may appear to be lacking in this constructor function is the setting of any default values for the 'Normal' member. The vertex normal is declared using the 'CVector3' class however, which defaults the X, Y and Z components to zero automatically within its own default constructor. The normal component's constructor will be called implicitly when an instance of the 'CVertex' class is created at any point within the application and as a result we need node duplicate this functionality here.

Of course, the condition under which the default constructor is called is only one of the many situations in which we may want to construct a vertex object. In many of our previous lab projects we have included several vertex class constructors which allow us to initialize its values quickly and efficiently. Similar alternate constructors have also been included in this application for the same reason.

```
    CVertex( float _x, float _y, float _z )
    {
        x = _x; y = _y; z = _z; tu = tv = 0.0;

    } // End Constructor
```

This constructor should be relatively self explanatory. It accepts three parameters which are simply used to initialize the positional portion of the vertex component. We can use this constructor in several situations including either object initialization or assignment as shown below:

```
// Stack Allocation / Initialization
CVertex Vertex1( 10.0f, 5.0f, 0.0f );

// Heap Allocation / Initialization
CVertex * pVertex = new CVertex( 10.0f, 5.0f, 0.0f ) ;

// Assignment
CVertex Vertex2 = CVertex( 10.0f, 5.0f, 0.0f ) ;
```

The other vertex components, such as the vertex normal and texture coordinate values, are set to an initial value of zero in this case.

The next constructor provided by this class is one that accepts a parameter of the type ‘CVector3’ which is again used to construct only the positional component of the vertex.

```
    CVertex( const CVector3& vec )
    {
        x = vec.x; y = vec.y; z = vec.z; tu = tv = 0.0f;

    } // End Constructor
```

This constructor is one of the most important for providing a level of interoperability between the new vector and vertex classes. As with the previous constructor we can use this for both initialization and assignment of the vertex data as shown below:

```
// Vector
CVector3 SomeVector( 10.0f, 5.0f, 0.0f );

// Stack Allocation / Initialization
CVertex Vertex1( SomeVector );

// Heap Allocation / Initialization
CVertex * pVertex = new CVertex( SomeVector );

// Assignment
CVertex Vertex2 = CVector3( 10.0f, 5.0f, 0.0f );
```

```
CVertex Vertex3 = SomeVector;
CVertex Result  = SomeVector.Cross( CVector3( 0.0f, 1.0f, 0.0f ) );
```

The latter of these examples – in which we assign the value of a vector directly to the vertex – is probably the most common situation in which this constructor will be utilized. The bottom-most assignment example shows how we might return the result from a vector cross product operation and store it directly within a vertex object. In this case, the vector assignment constructor will be called and the positional components updated. The remaining vertex components are initialized to a default of zero as before.

There are obviously several other components within our vertex class that may need to be initialized along with the vertex position. In response to this our vertex class also includes constructors which accept additional parameters for this purpose.

```
    CVertex( float _x, float _y, float _z, float _tu, float _tv )
    {
        x = _x; y = _y; z = _z; tu = _tu; tv = _tv;

    } // End Constructor

    CVertex( float _x, float _y, float _z, const CVector3& _Normal,
             float _tu, float _tv )
    {
        x = _x; y = _y; z = _z; Normal = _Normal; tu = _tu; tv = _tv;

    } // End Constructor
};
```

The final two constructors in our vertex class provide the ability to initialize the remaining vertex data not covered by those previously discussed. In each case there are additional parameters which match up to the data members exposed by the vertex. In the former of these final constructors, two new parameters labelled '_tu' and '_tv' are included. These will be used in circumstances in which the application would like to initialize the vertex using both positional and texture coordinate information. The latter of these two extends the first by including a parameter labelled '_Normal' which is used for the initialization of the vertex normal component.

There are of course many additional combinations of vertex class constructor which we may want to include at some point in the future. However, the constructors included here should be more than enough to provide our application with a flexible and comprehensive system by which vertices can be created, initialized and manipulated in many situations.

The CPolygon Class

With our vertex class fully defined, we now need a container in which they can be stored. As has been the case in many of the lab projects developed in previous lessons, the storage and logical grouping of the scene vertex information is the responsibility of the 'CPolygon' class. Although in some cases we have stored vertices directly within a mesh object – such as a 'CTriMesh' – this was almost exclusively built as a final step prior to its use within the rendering portion of the application.

As has been discussed in previous chapters, the spatial hierarchy compilation process that has been developed to date requires that the polygons used during the spatial partitioning phase are each convex winding ‘n-gons’. This is due to the fact that we often needed to split the polygon data, a process which is most easily achieved when the polygon data fits these criteria. In the case of the *polygon aligned* BSP tree however, the polygon concept plays a much more crucial role. Because the scene data itself will be used as the basis of the separating planes and nodes within the spatial hierarchy, it is vital that we ensure that each of the individual polygons within the scene remain separate. If we were to store all of the vertices directly within a mesh type container, indexing them using a triangle list for instance, this would obviously make the construction of our polygon aligned BSP tree much more complex.

Unlike these previous applications, we are now attempting to construct a much more comprehensive tool. In the past, the polygon class was often merely a means to an end in which vertex data was stored until it was used in the construction of hardware friendly vertex and index buffers for the purposes of being rendered. Our pre-processing tool on the other hand will need to work much more closely with various types of polygon data. As we add more features to the tool, there will be an increasing number of situations in which a particular compilation process will need to have access to additional polygon member variables, or even require an entirely custom polygon data structure. To see why this is the case we need only think back to the discussion of collecting polygons at the leaves in a polygon aligned BSP leaf tree.

At each stage in this compilation process, a polygon is chosen to become the separating plane for the current node in the hierarchy structure. Whenever a polygon is selected, it is removed from further consideration but it should still be passed down the relevant side in order to be collected in a leaf structure at a later stage in the process. This can be achieved very simply by exposing a custom ‘UsedAsSplitter’ flag from within the polygon itself for instance which is set whenever a polygon is chosen to become a node plane. As the already selected polygon is passed through the many levels of the recursive procedure, this flag can then be tested very quickly in order to determine if it should be used as a separating plane or not.

This is a prime example of a situation in which an individual compilation process might need an additional data member within the polygon structure. As you might imagine, there could be many circumstances under which this situation might arise as we add additional compiler modules to this tool. In the next chapter we will be introducing an additional compiler step that creates polygon-like constructs called portals. These portals are generated such that they describe the gaps in the geometry that exist between the leaf areas in the compiled tree. If you imagine that we had two rooms connected by a short length of corridor, the portal can be thought of as describing the holes that exist in each ‘doorway’ that connects each of the rooms to either side of that corridor. Essentially this will provide us with a form of leaf connectivity information that describes how each area of the scene connects to its one or more neighboring areas. In the case of the portal, we are only really interested in the shape of these portal polygons as defined by the vertices stored within them. In addition, this portal will need to store information about which leaf exists on either side of that portal polygon.

In the former of these example situations, we are required to add an additional member to the polygon class in order to store the ‘UsedAsSplitter’ information. With the later implementation of our portal compiler, there are yet more data members that are required and even some common concepts that

should not be included – such as texturing and material information – to keep memory requirements as low as possible. So, with the many types of polygon data requirements imposed upon us by this modular compiler concept, what is the best way to define our polygon class?

There are several ways in which this can be achieved. The first, and by far the simplest, is to add everything that we might need to our 'CPolygon' class, for every process that we implement, and simply not worry about any of the memory implications at all. While this certainly works in the majority of situations, it does mean that we will probably have to modify the polygon class every time we plug a new module into the compilation process. If our application makes copies of polygon members for any reason (such as during the splitting of that polygon into two new fragments), we will also have to ensure that the new members get duplicated across the board. As you might imagine, this could become a significant task and is somewhat in opposition to our wish for a modular pre-processing tool. Another way by which this can be achieved is for us to create a new polygon class for each process in which we need a custom data structure. This however, would require that much of the functionality which makes use of the polygon data and that which can be exposed by the polygon class itself – such as splitting, classification etc – may have to be duplicated for each type of polygon class that is implemented.

In this application we will be using a combination of both of these solutions in addition to borrowing one of the ideas we recently introduced in the new vertex class – that of utilizing class inheritance to make the process of development and future upgrades a little easier.

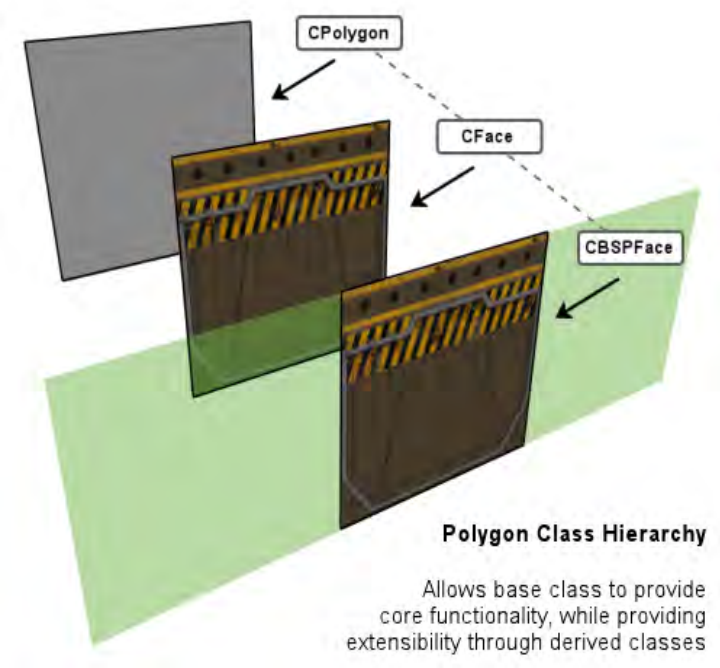


Figure 16.3

Figure 16.3 demonstrates how this type of hierarchical class design might work. In the top left of the image we can see our 'CPolygon' class which represents a plain polygon structure. In this class we would store only the vertex data required to define the polygon. This class would also implement several pieces of key polygon functionality such as the vertex storage and access functions, as well as the 'Split' function required by many of the spatial partitioning compilation techniques that we have already encountered. This base 'CPolygon' class would not declare any additional member variables or functionality that deal with additional surface properties such as texture or material information. By doing this we will later allow new structures to be derived from 'CPolygon' – such as the portal construct described earlier – that does not require any of these additional members, but does require the base functionality. With each of the vertex data members and management functionality provided by this base class, we can then derive a new class from 'CPolygon' in order to extend it for use in situations where additional rendering properties are required. This extended class is depicted in figure 16.3 by the polygon labelled 'CFace'. This class will include several additional

properties that we will need to store. Such properties might include texture, material and shader properties, surface normal, blending modes and any other additional data that we may want to use for rendering purposes in our runtime application. The final class in this diagram is depicted by the polygon labelled 'CBSPFace'. As you may have already guessed, this class is designed for use by our new BSP compiler during the spatial partitioning process. Because the BSP compiler will need to work directly with the *renderable* data, it makes sense that the new 'CBSPFace' class derives directly from the middle class 'CFace'. In this way, the custom BSP compiler polygon class will inherit any and all data members provided by its parent. It also allows us to extend this polygon class with any additional properties – such as our 'UsedAsSplitter' flag – that may be required **only** during BSP compilation.

We will discuss each of these classes in detail a little later on in this lesson. It is important however to understand the general design of our polygon data classes before we move on. Hopefully it is clear that by using a class hierarchy in this way, we are able to extend the functionality of the polygon class without the base class having to be aware of any of the additional properties required by any of the future processor modules. Each of these modules can simply derive a new class from the relevant base implementation which will automatically inherit the functions and data members defined by its parent thus sparing us from having to duplicate large amounts of functionality in each process specific polygon class.

With these general design concepts understood, let us move on to discussing the base 'CPolygon' class that is used within our new pre-processor tool.

```
class CPolygon
{
public:
    // Constructors & Destructors
             CPolygon( );
    virtual ~CPolygon( );

    // Public Variables for This Class
    CVertex        *Vertices;          // Polygon vertices
    unsigned long   VertexCount;       // Vertices in this poly

    // Public Virtual Functions for This Class
    virtual HRESULT Split( const CPlane3& Plane, CPolygon * FrontSplit,
                           CPolygon * BackSplit, bool bReturnNoSplit = false );

    // ** Non-Virtual Public Member Functions Omitted **

};
```

> **Note:** As with most class declarations shown throughout this workbook, member functions have been omitted where appropriate. Each of these omitted member functions is discussed in detail independently within that section.

As you can see, the class declaration for our base 'CPolygon' is relatively simple compared to those we have encountered in the past. We previously discussed how our base 'CPolygon' class would only contain the basic member variables and functionality required to define the polygon layout itself. This is in order to allow any individual compiler module to derive from the most appropriate base class. As a result, this declaration includes only two member variables.

CVertex *Vertices

The first and most important member variable declared within this class is a pointer that will store the address to our heap allocated linear vertex array. Each element in this array will contain a 'CVertex' object in the same way as we have seen implemented many times before throughout this course. Each of the vertices in this array defines the shape and boundaries of the convex winding 'n-gon' that is a pre-requisite for the compiler procedures that we will go on to implement shortly. Remember that our 'CVertex' class is derived from 'CVector3' and as a result we will be able to perform 3D vector operations directly on any element within this array.

unsigned long VertexCount

This member variable contains the value which describes the total number of vertices stored within the above vertex array. Although we have become accustomed to using STL vectors for the storage of data within recent lab projects – which would remove the need for a separate count variable – the vertex array stored here is most often static in size. As a result it is sometimes more efficient for us to handle the array allocation / resizing manually in this way.

The final important point to notice about the base 'CPolygon' class is how the 'Split' function has been declared. Although this function and its parameters should be very familiar to you at this stage, unlike the previous implementations this function has been declared as *virtual*. This allows us to safely include a new split function in any future polygon class in order to extend it with additional functionality. In addition, the class destructor is also declared as virtual to ensure that when the application deletes an instance of a polygon using a pointer of a *base* class type, the data members declared by the whole class hierarchy are correctly released. We will see how these concepts can benefit us when we move on to discuss the derived 'CFace' class in the next section.

CPolygon::CPolygon()

In this application, the default 'CPolygon' constructor does not have a great deal of work to do. As demonstrated in the following code, this constructor simply initializes the two base class member variables to ensure that they can be tested and used safely in later functions.

```
CPolygon::CPolygon()
{
    // Initialise anything we need
    Vertices    = NULL;
    VertexCount = 0;
}
```

CPolygon::AddVertices

This function allows the application to reserve space within the polygon for the required number of vertices. It can be called multiple times in order to grow the vertex array in each subsequent call. In each case, this function will return the index to the first vertex added to the array in that specific call. Although we have encountered this type of function several times in past lessons, we are using a different vertex concept in addition to a slight variation of the error handling concepts we are familiar with.

```
long CPolygon::AddVertices( unsigned long nVertexCount )
{
    CVertex *VertexBuffer = NULL;

    // Validate Requirements
    if ( nVertexCount == 0 ) return -1;
```

The only parameter accepted by this function allows the application to specify how many additional vertices we would like to add to the vertex array stored here. If the vertex array has already been created by a previous call, this will be the number of elements by which that array will be grown. As in the many functions we have implemented before, this parameter is validated to ensure that the allocation will not fail due to the application specifying a count of zero.

The first real task that this function must undertake is to allocate a temporary array of the correct size that will be used as the new vertex array.

```
    // Allocate brand new buffer
    try
    {
        VertexBuffer = new CVertex[ VertexCount + nVertexCount ];
        if (!VertexBuffer) throw std::bad_alloc(); // VC++ Compat
```

Here we allocate the memory that will be used to build a new vertex array of the requested size. We specify a number of vertices to be allocated which is made up of the number of additional vertices requested by the application (nVertexCount) added to the number of vertices that were previously stored within the polygon. This array of vertices will eventually replace the one currently stored within the polygon. Remember that because the 'CVertex' class contains a default constructor, there is no need for us to initialize the elements within the vertex array as we may have done in the past.

In previous implementations of the memory allocation functions, we have often simply tested the resulting value stored within the temporary buffer variable to see if it contains a 'NULL' pointer. In some version of Microsoft Visual C++ this was the means by which the 'new' operator signalled that an allocation failure had occurred. In later versions of Visual C++ however this behaviour was changed such that an exception will be thrown in this case. In order for us to support both methods of error detection we have wrapped the allocation of the new vertex array in a 'try' block. This ensures that if the 'new' operator throws an exception, we are able to handle this failure gracefully. To enable us to provide backwards compatibility however, we have included the aforementioned 'NULL' pointer test here also. In this case, we generate the same exception as we would expect the 'new' operator to throw. In doing so, we are able to support both types of error handling simply by including a single catch block.

```
        // If any old data
        if (Vertices)
        {
            // Copy over old data
            memcpy( VertexBuffer, Vertices, VertexCount * sizeof(CVertex));

            // Release the memory allocated for the original vertex array
            delete []Vertices;
```

```
        } // End if old vertices created

    } // End try block
```

This next portion of the code tests to determine whether or not vertex data already exists within this polygon. If this is found to be the case, then that vertex data is copied into the new vertex array to ensure that any vertices already inserted by the application persist. Notice how we copy only the amount of data that is contained within the *original* vertex array. This is important because we need to ensure that we do not attempt to read any memory past the end of the previously allocated block. This means that if our polygon previously contained 5 vertices, and our application was requesting space for an additional 5 then only those original vertices would be copied into the first 5 elements of the new array, leaving the last 5 elements in their default state.

The final task in this block of code is to release the memory (if any) that was allocated in a previous call to this function. If a vertex array already exists, we must release it to ensure that our application does not leak any memory when the 'Vertices' member pointer is overwritten with that of our new array.

With all of our memory allocation and initialization completed, there are no further exceptions which are likely to be generated within the context of this function.

```
    // Was an exception thrown?
    catch (...)
    {
        if (VertexBuffer) delete []VertexBuffer;
        return -1;

    } // End catch block
```

The catch block we are using in this function specifies an ellipsis in place of a specific exception handling type. What this basically means is that this catch block will capture any and all exceptions generated in the try block regardless of the type of exception thrown. Because the application is not interested in the specifics of any exception that may have occurred in this function, we simply release the memory allocated for the new vertex array – if the exception was thrown after the allocation line – and return a value of -1. Because the application is expecting a valid index in the range of 0 and above, this *invalid* index return code can be used to inform the calling function that the allocation was **not** successful.

At this point we have allocated a new vertex array that has enough elements to contain all of the original data that was stored in the array prior to this call, as well as those additional elements requested by the calling function. With the original data back into this new array, and the original one released, we can now simply overwrite the old vertex array pointer, with that of the new array. We also increment the polygon's internal 'VertexCount' member variable to ensure that both the application and subsequent calls to this function have access to up to date information about the size of the vertex array.

```
    // Store the new buffer
    Vertices     = VertexBuffer;
```

```
    // Increment vertex count
    VertexCount += nVertexCount;
```

The final task that must be undertaken here is to return the index to the first of the vertex elements added to the array during that call. This index will be equal to the number of vertices that were originally stored in the polygon before this function was called. Returning the correct index can be achieved in one of two ways, either by making a temporary copy of the 'VertexCount' variable before we increment it, or by subtracting the requested additional vertex count parameter from the newly incremented 'VertexCount' value. Our implementation chooses the latter of these two methods as shown in the following snippet:

```
    // Return the base vertex
    return VertexCount - nVertexCount;
}
```

Although there is a relatively small amount of code in this function, it was important to discuss this given the changes in the way we are handling errors within our pre-processing tool. This exception handling scheme is used throughout this new application, and as a result we will not be going into quite as much detail about future memory allocation routines.

CPolygon::InsertVertex

There are relatively few cases in which we will ever need to insert a vertex into a polygon after it has been built. In most cases we will be generating new polygons with which we would build new vertex arrays from scratch each time. With that said, there are situations in which certain tasks are made easier by inserting a single vertex into a particular position around the polygons exterior. An example of such a case is with the recently covered T-Junction repair process. Because we have covered this function recently, we will only provide a brief recap to demonstrate this function as it applies to our new tool.

As before, this function's main purpose is to reshuffle the polygon's internal vertex array in order to create space for a new vertex. This vertex will be inserted *before* the vertex specified by the single parameter passed to this function. By doing so, we ensure that the new vertex will have an index that is equal to the value requested by the calling function.

```
long CPolygon::InsertVertex( unsigned long nVertexPos )
{
    CVertex *VertexBuffer = NULL;

    // Add a vertex to the end
    if ( AddVertices( 1 ) < 0 ) return -1;
```

The first thing we need to do here is to resize our vertex array to make sure that here is enough room for a single new vertex to be inserted. As outlined in our earlier coverage, the 'AddVertices' function will return a value of -1 if the allocation was unsuccessful. If this is found to be the case, then this function will return immediately using a similar error code.

At this stage, the polygon remains unaltered with the exception that there is a new empty vertex at the end of the internal vertex array. Due to the fact that our application requires a free slot to be created at an arbitrary position within the array – not necessarily at the end – we must shuffle the array elements such that this spare vertex now exists at the correct location. This is achieved by moving each of the vertices along by one, starting with the final element from the original vertex array, continuing backwards until we have moved each vertex up to and including the element specified by the input parameter. This could be achieved using the following code:

```
long i; // Signed type, we're counting backwards
for ( i = VertexCount - 2; i >= nVertexPos; ++i )
{
    // Move the current vertex forward by one.
    Vertices[i + 1] = Vertices[i];

} // Next Vertex
```

In this code you can see that we are simply assigning each vertex to the value of the one that falls directly prior to it in the vertex array. This is repeated until we reach the requested position, leaving a gap in the array at the element on which the loop exits.

Although this is a relatively simple loop, there is a quick and easy way to achieve this same effect using the 'memmove' function exported by the standard C runtime library.

```
    // Reshuffle the array unlesss we have inserted at the end
    if ( nVertexPos != VertexCount - 1 )
    {
        // Move all the verts after the insert location, up a ways.
        memmove( &Vertices[nVertexPos + 1], &Vertices[nVertexPos],
                 ((VertexCount-1) - nVertexPos) *  sizeof(CVertex) );

    } // End if not at end.
```

Internally, this function works in a similar fashion to the looping method described previously. The 'memmove' function is not however concerned with the layout and structure of the data or any form of class assignment or protection. It basically performs a byte for byte transfer of the specified area of memory, into the required destination address. This is not a problem for our vertices because they are essentially just data structures that do not use any form of virtual function mapping that may cause problems when moving the array contents in this way. One test that we must perform before we call this function however is to determine whether the requested vertex position is already at the end of the array. If this was found to be the case, then the data area that we would pass to this function would be empty and could potentially cause the operation to fail. Since the previous call to the 'AddVertices' function results in the new vertex being added to the end of the array, we need take no further action if this was the position requested by the calling function.

Once the contents of the array have been moved such that a gap remains at the correct location, we must finally overwrite the values of that spare vertex with those of a default state and return a success code to the calling function.

```
    // Initialize data to default values
```

```
    Vertices[ nVertexPos ] = CVertex( 0.0f, 0.0f, 0.0f );

    // Return the position
    return nVertexPos;
}
```

CPolygon::Split

The polygon split function remains largely unchanged from the implementations we have seen in earlier lab projects. However, there are a few alterations that have been made to allow us to take full advantage of our new hierarchical polygon class design.

```
HRESULT CPolygon::Split( const CPlane3& Plane, CPolygon * FrontSplit,
                         CPolygon * BackSplit, bool bReturnNoSplit )
{
    ...

    // Local variable definitions remain unchanged

    ...

    // New local variable
    CVertex NewVert;

    ...

    // Temporary array allocation and early-out logic is
    // largely unchanged. As a result it is not included here.

    ...
```

Although this function is largely identical to the previous implementation, there are some changes that affect how the ‘FrontSplit’ and ‘BackSplit’ parameters are declared and how they should be used by the application. Up until this point, the split function has been responsible for allocating any new polygon instances in cases where the source polygon is found to be spanning the input plane. These new polygon instances were returned to the calling function via the ‘FrontSplit’ and ‘BackSplit’ parameters which were previously declared as double ‘CPolygon’ pointers:

```
Split( const D3DXPLANE& Plane,
       CPolygon ** FrontSplit, CPolygon ** BackSplit, bool bReturnNoSplit )
```

Recall that our application would pass in a pointer to another ‘CPolygon’ pointer *variable* that would then be dereferenced by the split function in order to store the new polygon pointers in the variables referenced by these two parameters.

In this implementation we do not wan to force the application to use any particular polygon class at any point in the process. If the split function was to allocate any new polygon objects using the ‘CPolygon’ type, then we would be unable to make use of any form of class hierarchy in this way. As a result, the calling function must now be responsible for creating the polygon instances in advance – using the

applicable class type – and pass in these already instantiated objects via these same two parameters. Because we are no longer passing in pointers to variables as before, these two parameters are now simply single 'CPolygon' pointers. Although it may seem as though we are still placing a restriction on the application by accepting pointers of this base class type, remember that in C++ we are able to use a pointer to any type found within the class hierarchy when we need to access the superclass information. In the case of this base split function, we will only be modifying the properties defined by the base 'CPolygon' class itself.

We mentioned in the comments included in the previous code snippet that all of the logic we have previously implemented for the temporary array allocation and any 'early out' testing remains the same. Let us therefore move on to discussing those changes made to the core splitting functionality.

```
    // Compute the split if there are verts both in front and behind
    if (InFront && Behind)
    {
        for ( i = 0; i < VertexCount; i++)
        {
            // Store Current vertex remembering to MOD with number of vertices.
            CurrentVertex = (i+1) % VertexCount;

            if (PointLocation[i] == CLASSIFY_ONPLANE )
            {
                if (FrontList) FrontList[FrontCounter++] = Vertices[i];
                if (BackList)  BackList [BackCounter ++] = Vertices[i];
                continue; // Skip to next vertex
            } // End if On Plane

            if (PointLocation[i] == CLASSIFY_INFRONT )
            {
                if (FrontList) FrontList[FrontCounter++] = Vertices[i];
            }
            else
            {
                if (BackList) BackList[BackCounter++] = Vertices[i];

            } // End if In front or otherwise

            // If the next vertex is not causing us to span the plane then continue
            if ( PointLocation[CurrentVertex] == CLASSIFY_ONPLANE ||
                 PointLocation[CurrentVertex] == PointLocation[i]) continue;

            // Calculate the intersection point
            Plane.GetRayIntersect( Vertices[i], Vertices[CurrentVertex],
                                   NewVert, &fDelta );
```

The above code also remains largely unchanged. The only real difference here is that we are no longer using our collision library to retrieve information about the relative locations of each vertex in respect to the specified plane. Notice the last line in this snippet in which we are using one of our new 'CPlane3' class functions named 'GetRayIntersect'. This is in place of the previous call to the 'CCollision' class's 'RayIntersectPlane' function that has not been included in this application. Unlike the previous ray

intersection function – which required a ray start point and a ray velocity – this function accepts both a ray start and end point as its input parameters. Notice however that we are passing the vertices directly into this function without any sort of casting, even though the function we are calling is declared using 'CVector3' parameter types. This is just one of the benefits of deriving our vertex from the vector class as discussed earlier.

The 'GetRayIntersect' function outputs both the point in space at which the ray intersects the plane, as well as the optional 't' value we are familiar with. In the initial code listing for this function you may have noticed that we included a new local variable named 'NewVert'. This variable was also of the type 'CVertex'. Once again, because of our vertex class hierarchy design, the intersection function is able to set up the vertex values directly even though it is expecting a parameter of the type 'CVector3'. We also pass to this function the floating point variable 'fDelta', into which the intersection 't' value will be stored. We will need both of these values in order to correctly interpolate the remaining vertex components.

In this next block of code we perform these remaining vertex component interpolation steps. This ensures that after the polygon has been split, any new vertices inserted by the splitting procedure contain the correct values based on their position along the edge of the original polygon. Once the new vertex has been completely initialized, we then store it in both of the front and back fragment lists as before.

```
            // Interpolate Texture Coordinates
            CVector3 Delta;
            Delta.x    = Vertices[CurrentVertex].tu - Vertices[i].tu;
            Delta.y    = Vertices[CurrentVertex].tv - Vertices[i].tv;
            NewVert.tu = Vertices[i].tu + ( Delta.x * fDelta );
            NewVert.tv = Vertices[i].tv + ( Delta.y * fDelta );

            // Interpolate normal
            Delta          = Vertices[CurrentVertex].Normal - Vertices[i].Normal;
            NewVert.Normal = Vertices[i].Normal + (Delta * fDelta);
            NewVert.Normal.Normalize();

            // Store in both lists.
            if (BackList)  BackList[BackCounter++]   = NewVert;
            if (FrontList) FrontList[FrontCounter++] = NewVert;

        } // Next Vertex

    } // End if spanning
```

Once the front and back vertex lists have been created, we then store this data into the applicable polygon fragments allocated and passed in by the application. Unlike previous polygon class implementations however, this class does not store any additional information which needs to be duplicated into those fragments. This will in fact be handled by those classes which are derived from this base class. We will see how this achieved in the next section where we will discuss one of these derived classes.

```
    // Allocate front face
    if (FrontCounter && FrontSplit)
    {
```

```
        // Copy over the vertices into the new poly
        FrontSplit->AddVertices( FrontCounter );
        memcpy(FrontSplit->Vertices, FrontList, FrontCounter * sizeof(CVertex));

    } // End If

    // Allocate back face
    if (BackCounter && BackSplit)
    {
        // Copy over the vertices into the new poly
        BackSplit->AddVertices( BackCounter );
        memcpy(BackSplit->Vertices, BackList, BackCounter * sizeof(CVertex));

    } // End If
```

With our polygon fragments fully constructed, the only thing that remains for us to do is to clean up any of the temporary arrays that may have been allocated at the beginning of the function, and return our success code.

```
    // Clean up
    if (FrontList)      delete []FrontList;
    if (BackList)       delete []BackList;
    if (PointLocation)  delete []PointLocation;

    // Success!!
    return BC_OK;
}
```

CPolygon:: ReleaseVertices / ~CPolygon()

The final two base polygon member functions are those that are responsible for cleaning up and releasing any memory allocated and stored within this class. The first of these is the 'ReleaseVertices' function. This is a public member function that can be called by the application if it needs to release the memory allocated by this polygon, perhaps in order for this object instance to be reused. The second of these functions is the class destructor. Rather than duplicate the clean up code provided by the 'ReleaseVertices' function, the class destructor simply calls it at the point where the polygon object is released or goes out of scope.

```
void CPolygon::ReleaseVertices()
{
    // Clean up after ourselves
    if ( Vertices ) { delete []Vertices; Vertices = NULL; }
    VertexCount = 0;
}
```

```
CPolygon::~CPolygon()
{
    // Clean up after ourselves
    ReleaseVertices();
}
```

With our polygon base class defined, let us now take a look at how we might derive a new polygon class that will provide the data members needed to store any renderable polygon information loaded from file.

The CFace Class

This is the first of our derived polygon classes. It is used by a large portion of our application for the storage of renderable polygon information. That is the polygon data which is loaded from file, compiled and exported, and is finally intended to be rendered by our game application. As a result, this class stores the common pieces of information we have come to rely on such as the polygon normal, texture and material information in addition to any flags and alpha blending properties that may be required.

Recall during our discussion of the new polygon class hierarchy design, we ideally want to create a class structure in which each module is responsible for defining its own variables within an independent polygon class. However, at this point in time we need an intermediate concept that we can use to load the data contained within the source IWF file. This will enable us to initialize each processing module easily, using a common polygon structure, from which they can each base their own storage classes. At a later point in this lesson, we will derive a further class from this one which will be responsible for storing additional information required during the BSP compilation process for instance.

Due to the fact that this class is derived from the 'CPolygon' base class, much of the functionality is inherited. As a result, we are only really responsible for extending this class for the purposes of storing and retrieving any additional render based information. Let us therefore take a look at the declaration for this intermediate storage class.

```
class CFace : public CPolygon
{
public:
    // Constructors & Destructors
    CFace( );

    // Public Variables for This Class
    CVector3        Normal;                 // Face Normal
    short           TextureIndex;           // Index into texture look up
    short           MaterialIndex;          // Index into material look up
    short           ShaderIndex;            // Index into the shader look up
    ULONG           Flags;                  // Face Flags
    UCHAR           SrcBlendMode;           // Face Source Blend Mode
    UCHAR           DestBlendMode;          // Face Dest Blend Mode

    // Public Virtual Functions for This Class
    virtual HRESULT Split( const CPlane3& Plane, CFace * FrontSplit,
                           CFace * BackSplit, bool bReturnNoSplit = false );

};
```

Earlier we mentioned that any applicable member functions have been omitted from many of the class declarations being discussed throughout this chapter. In the case of this derived class however, with the exception of the virtual 'Split' function shown, this class defines no additional member functions. Much of the functionality available to the application through an instance of this type is provided by the base

'CPolygon' class from which it inherits. There are however several new member variables declared here that are discussed below.

CVector3 Normal
This member variable is responsible for storing the normal vector for this polygon / surface. As you should already be aware, this piece of information is extremely important to us when we are constructing a *polygon aligned* BSP tree compiler. This is due to the fact that both the polygon normal, and at least one of its vertices, is used as the basis for creating the separating planes assigned to each node in the tree.

short TextureIndex
Previously we mentioned that this class is responsible for storing each of the pieces of information that may be required to render this polygon in our final application. As a result, we store the index into a texture information array in this member variable. Although this application does not perform any sort of rendering itself, the texturing information must be exported back out to file when the compilation process has been completed. Notice that we declare this member variable using a signed type. This is because we use a value '-1' to signify that no texture has been assigned to this particular polygon.

short MaterialIndex
As with the aforementioned 'TextureIndex', we also store an index into the global material array much as we have done before in previous lab projects. Again this is a signed type in which '-1' signifies that no material has been applied to this polygon and a default material should be assumed. Although we have not yet discussed the application texture and material arrays at this point, we will see later how this information is loaded from file within our main compiler class.

short ShaderIndex
At this point in the course we have not yet encountered the use of pixel or vertex shaders within our application. However, we are currently constructing a tool which will hopefully be applicable to future lessons. As a result we also retain additional pieces of information that are available in the source IWF file such as the per-polygon shader index. As with each of the texture and material indices, a value of '-1' signifies that no shader has been applied to this polygon.

ULONG Flags
As we have discussed in previous lessons, each of the surfaces stored within the IWF file also carries with it a number of possible flags. These include as an example whether the polygon should be considered invisible or not. Although we do not pay an attention to this information within the various compilation processes in this application, the flags assigned to each polygon should be retained in order for them to be exported back out to the compiled file.

UCHAR SrcBlendMode
The source alpha blending mode is again another piece of information that is only really applicable to our rendering application. This information is loaded from file and retained here only for the purposes of exporting it back out to file for the run-time application's import procedure to process.

UCHAR DestBlendMode

This member variable is the complement of the source blending mode variable and is provided only for the purposes of export for the rendering application.

Although many of these member variables serve no other purpose than to be exported to the resulting file, it is important that this information is retained and is processed as each compiler task is performed. During the BSP compilation process for instance, we already know that many polygons will be discarded and likewise new polygons created in their place. As a result it is imperative this information be considered in several places in the application and – as we will discover shortly – one such place is during the splitting of our polygon data against the separating planes.

This class is of course merely an extension of the base 'CPolygon' class we covered in great detail in the previous section. As a result, there are only two member functions exposed by this class, each of which we will be discussing next.

CFace::CFace()

As with most of our classes, we start with the class constructor. We have defined several new member variables in this derived class that we need to initialize. Remember that in C++, the constructor of the base classes are called automatically when an object of this type is created. As a result we need only initialize the variables that are declared explicitly by this class declaration.

```
CFace::CFace()
{
    // Initialise anything we need
    MaterialIndex = -1;
    TextureIndex  = -1;
    ShaderIndex   = -1;
    Flags         =  0;
    SrcBlendMode  =  0;
    DestBlendMode =  0;
}
```

CFace::Split

Here is where our new hierarchical polygon class design comes into its own. Even though we have declared many new member variables that need to be duplicated into each of the split fragments, we need not re-implement the split functionality provided by the base 'CPolygon' class.

```
HRESULT CFace::Split( const CPlane3& Plane, CFace * FrontSplit,
                      CFace * BackSplit, bool bReturnNoSplit )
{
```

As before, the first parameter to this derived 'Split' function is a reference to the separating plane that will be used to classify and potentially split the polygon into two component fragments. The second and third parameters are pointers to each of the *pre-allocated* polygon instances into which the split function will place the output data. Notice however that the types of these two polygon pointers are declared as 'CFace', rather than 'CPolygon' as before. This allows us to both inform the application about the type of polygon this function is expecting, as well as granting us easy access to the newly defined member

variables within this class. This is why it was vital that we made the alterations discussed earlier, such that the application would pass in pointers to two pre-allocated polygon objects of the correct type. Because we are being passed two new 'CFace' objects via the 'FrontSplit' and 'BackSplit' parameters, we will be guaranteed to have access to the new member variables that we have declared.

The fourth and final parameter is the early out flag that informs the split procedure whether or not to return immediately if no split has occurred. Recall that if the application passes true to this parameter and the polygon's vertices are found not to span this plane, then the function will exit immediately without copying any information into either of the child fragments. If the application should specify false, then this function would proceed to copy the vertex data into the appropriate fragment even if no split has occurred.

Due to the fact that a large portion of the tasks undertaken during the splitting of a polygon are related to the building and separating of vertex data, there is obviously a lot of common functionality that we have already implemented in the base 'CPolygon'. As a direct result of deriving the 'CFace' class from 'CPolygon' in this way, we are able to make use of this functionality by simply calling the base class' 'Split' function, passing in each of the parameters specified by the application directly:

```
    // Call base class implementation
    HRESULT ErrCode = CPolygon::Split( Plane, FrontSplit, BackSplit,
                                       bReturnNoSplit );
    if (FAILED(ErrCode)) return ErrCode;
```

At this point, the base class implementation of the split function will have separated any applicable vertices into the appropriate polygon fragments specified by the application. Of course the base implementation is not aware of any of the member variables that have been included in this extended polygon class. To this end, the only remaining task that our derived class's split function must undertake is to copy the data stored within each of these variables, into those of the new split fragments as shown in the following snippet.

```
    // Copy remaining values
    if (FrontSplit)
    {
        FrontSplit->Normal        = Normal;
        FrontSplit->MaterialIndex = MaterialIndex;
        FrontSplit->TextureIndex  = TextureIndex;
        FrontSplit->ShaderIndex   = ShaderIndex;
        FrontSplit->Flags         = Flags;
        FrontSplit->SrcBlendMode  = SrcBlendMode;
        FrontSplit->DestBlendMode = DestBlendMode;
    } // End If

    if (BackSplit)
    {
        BackSplit->Normal         = Normal;
        BackSplit->MaterialIndex  = MaterialIndex;
        BackSplit->TextureIndex   = TextureIndex;
        BackSplit->ShaderIndex    = ShaderIndex;
        BackSplit->Flags          = Flags;
```

```
        BackSplit->SrcBlendMode   = SrcBlendMode;
        BackSplit->DestBlendMode  = DestBlendMode;
    } // End If

    // Success
    return BC_OK;
}
```

You can hopefully gather from this simple function that: in creating a relatively simple class hierarchy design, we are able to easily extend the functionality of the base class and provide new custom member variables at will. This will be of enormous benefit to each of the future compilation process modules that we will construct because it allows them to create their own polygon structures and store custom information without having to re-implement any of the common polygon functionality.

One other item of note is that we have not included a destructor in this class. Remember that because this class is derived from 'CPolygon', this base class destructor will always be called. Due to the fact that none of the member variables in the extended 'CFace' class contain pointers to any form of allocated memory we do not need to include any release or cleanup code. The vertices that were allocated will however be released by the base class constructor automatically, even if we were to include a destructor here.

With the 'CFace' class fully defined, we are now able to load and save the renderable polygon information to and from the source and destination IWF files. However, we still need to implement a class that serves as the container for this polygon data.

The CMesh Class

The mesh class is a concept that will already be very familiar to us. We have developed several different mesh classes in previous lab projects, but these too were heavily biased toward the construction of hardware friendly buffers used for rendering. As with the polygon classes we have covered so far, the mesh class we are developing in this tool is intended to store information only for the purposes of easy access to the scene data and final export to file.

```
class CMesh
{
public:
    // Constructors & Destructors
             CMesh( );
    virtual ~CMesh( );

    // Public Variables for This Class
    char           *Name;                // Stored name loaded from IWF
    CFace         **Faces;               // Mesh faces
    ULONG           FaceCount;           // Faces in this mesh
    CMatrix4        Matrix;              // Meshes object matrix
    CBounds3        Bounds;              // Meshes local space bounding box
    ULONG           Flags;               // Mesh flags (i.e. detail object)

    // Public Member Functions Omitted
```

```
};
```

This mesh class is relatively simple and declares only a handful of member variables. These variables match the information stored within the IWF file. Let us examine each of these in detail.

char * Name
Each of the meshes contained in the scene geometry file can have a name associated with them. This name string is usually specified by the artist or level designer within a scene editor such as GILES™ which can be used by the application to identify a particular mesh. As an example of why this might be useful, imagine that we created a mesh that served as a door within the level. Our application might want to open this door whenever the player triggered a script by entering the correct code on a door panel. Rather than opening every door in the level, this script would obviously need to identify a particular door to open. By naming each of the door meshes individually, the script writer would then be able to identify just the specific door that should open when the correct code has been entered into that panel.

Again this information is not used directly within our compilation tool, but we are required to retain this string in order to export it once again, after the compilation procedures have been executed.

CFace ** Faces
Because the mesh data stored in the file will ultimately be used for rendering by the final game application, this mesh class is designed to store and manage polygons using the 'CFace' class type. As a result, this member variable stores a pointer to an array of 'CFace' object instances rather than its base class 'CPolygon' found in previous implementations. Notice how we are using a double pointer declaration here. This is because we want to store an array of 'CFace' *pointers* which allows us to perform a simple shallow copy on the array each time it is resized. Just as our polygon class exposed a series of functions to manage the internal vertex array, the mesh class does the same in order to allow our application to manage this polygon array in an easy to use fashion.

ULONG FaceCount
This variable simply stores a value describing the number of polygons currently stored in the aforementioned 'Faces' array. We need to maintain this information in order to resize this array in each call to the 'AddFaces' member function, in addition to correctly reporting the length of the array to the application.

CMatrix4 Matrix
In previous applications, mesh object matrices were maintained by a wrapping 'CObject' class. This was provided in order to allow our application to create references to individual meshes and still maintain the ability to position each mesh independently. In the IWF file however, this reference information is maintained by individual 'entity' structures within the file and will be handled separately. In effect, each individual mesh may have its own matrix that describes the position and orientation of the *first* instance of that mesh. This first mesh may then be subsequently referenced by an entity that will override this position and orientation information. Again, because we are not rendering this mesh data, we have no need for the wrapping object class concept and therefore this matrix is simply maintained directly within the mesh itself. In GILES™, the vertices stored within each polygon are expressed in world space directly. This means that for the most part this member variable will contain an identity matrix. With that said, other scene creation tools may export an IWF file that does not define its mesh vertices in

world space. This imported variable should therefore still be taken into consideration. We will discuss how this matrix is used within the pre-processing tool a little later on in this lesson.

CBounds3 Bounds
This variable describes the axis aligned bounding box extents of the mesh vertex data in world space. The bounding box information provides us with a quick initial broad-phase mechanism by which we can test for intersection between two meshes. Among other things, this variable will be used later on in this chapter during the hidden surface removal process to determine whether two meshes might need to be merged together using the CSG union operation.

ULONG Flags
Just as with the polygon / surface information stored within the IWF file, each mesh also carries with it a number of possible flags. These include as an example whether the mesh itself is a detail object that should not be considered during the BSP compilation step. We will discuss the implications of these flags when they are encountered during the coverage of each of the main compilation tasks.

Let us now discuss the functions to the CMesh class that were omitted from the class declaration. We will start by looking at the initialization procedure (the constructor) and will then discuss the array management. Finally we will look at the various utility functions that allow us to more easily integrate the mesh into the compilation tool.

CMesh::CMesh()

The only constructor defined by the new mesh class is the default one, which simply initializes the member variables of the mesh whenever an instance of this class is created.

```
CMesh::CMesh()
{
    // Initialise anything we need
    Faces       = NULL;
    FaceCount   = 0;
    Flags       = 0;
    Name        = NULL;
}
```

As you can see, we simply set each of the member variables to a default state of either NULL or 0. In the case of the two pointer variables – ‘Faces’ and ‘Name’ – we set them to an initial state of NULL in order for us to identify whether or not any memory has already been allocated and should potentially be released. If either of these values is found to be equal to NULL at a later point, then we should obviously not attempt to release the memory to which this variable points because it had not yet been allocated.

CMesh::AddFaces

This is a public method that is used by the application in order to grow the ‘CFace’ array maintained by an instance of this mesh class by the specified amount. Lab project 16.2 uses this function when loading internal meshes within the IWF file in addition to cases where it is required to manually construct a new mesh as we will see shortly.

The ‘AddFaces’ function is similar in many respects to the memory allocation functions we have been using throughout the development of each lab project in this series. However, as with the polygon’s ‘AddVertices’ function, there are a number of key differences that we must draw your attention to.

```
long CMesh::AddFaces( unsigned long nFaceCount )
{
    CFace **FaceBuffer = NULL;

    // Validate Requirements
    if ( nFaceCount == 0 ) return -1;
```

The only parameter accepted by this function allows the application to specify how many additional polygons we would like to add to the ‘CFace’ array stored here. If the array has already been created by a previous call, this will be the number of elements by which that array will be grown. As in the many functions we have implemented before, this parameter is validated to ensure that the allocation will not fail due to the application specifying a count of zero.
This function has a return type of a simple signed long. This will be used to return the index to the first of the polygon elements added to the array during that call. This index – as always – will be equal to the number of polygons that were originally stored in this mesh before the ‘AddFaces’ function was called. If an error occurred during the execution of this function, a value of ‘-1’ will be returned.

As before, our first job is to attempt to allocate a new array of the same type as the internal ‘Faces’ member variable that has been correctly sized to take into account those new requested polygons. Subsequently, any data already stored within the mesh must then be copied into this new polygon array. Because we are simply storing *pointers* to heap allocated ‘CFace’ objects, this can be achieved by a single call to the standard C runtime ‘memcpy’ call. This new array will eventually be used to overwrite the one currently stored by the mesh.

```
    try
    {
        // Allocate brand new buffer
        FaceBuffer = new CFace*[ FaceCount + nFaceCount ];
        if (!FaceBuffer) throw std::bad_alloc(); // VC++ Compat
        ZeroMemory( FaceBuffer, (FaceCount + nFaceCount) * sizeof(CFace*) );

        // Copy over any old data
        if (Faces) memcpy( FaceBuffer, Faces, FaceCount * sizeof(CFace*));

        // Allocate new faces
        for ( long i = 0; i < (signed)nFaceCount; i++ )
        {
            FaceBuffer[FaceCount + i] = new CFace;
            if (!FaceBuffer[FaceCount + i]) throw std::bad_alloc(); // VC++ Compat

        } // Next face

    } // End Try Block
```

There is one very important addition that has been made to the above code that we have not yet encountered in any of our memory allocation routines. The latter part of the function actually allocates the new 'CFace' objects on behalf of the application and stores them in the array in advance. There are several reasons why we have chosen to do this with our new mesh class. The first of these is due to our new class hierarchy design. Because the application has much more freedom to create an object using one of the several potential types of polygon classes, we have introduced a situation in which the application could create a polygon object of a type which does not store much of the information required. This could also lead to a situation in which one of the virtual functions defined by the polygon class could be called erroneously. By allocating the polygon objects explicitly within this function, we are able to enforce the type of polygon stored within each element of the mesh's member array. An additional benefit of including this allocation step here is that we are able to make use of the error handling logic in this function without having to include that same logic each time we need to allocate a mesh polygon.

At this point in the function, we have allocated the newly sized array and copied over any of the polygon pointers previously stored in the 'Faces' member variable. A series of new 'CFace' class instances have also been created and initialized by their own constructors, and finally stored within this new array. As always, there are potentially many situations under which an error may have occurred with all of the memory allocation that has been performed.

All of the allocation steps within this function have been wrapped in a try block that will allow us to capture any exceptions that may have been thrown during the execution of the lines of code within that block. Backwards compatibility has also been preserved by testing the pointer value returned from each allocation operation and explicitly throwing a similar exception if the value was found to be 'NULL. Of course, an exception may be thrown at any point in this block of code that might result in only some of the intended allocations having been attempted. Therefore, in the event of an exception, we must clean up and release those memory areas that were successfully allocated. In the following catch block, a great deal of care is taken to test each of the potentially allocated memory pointers and release them only if they are *not* equal to NULL. This should ensure that we do not attempt to deallocate memory that was not allocated prior to the exception being thrown.

```
    // Did an exception get thrown ?
    catch (...)
    {
        if (FaceBuffer)
        {
            // Release any already allocated faces
            for ( long i = 0; i < (signed)nFaceCount; i++ )
            {
                if (FaceBuffer[FaceCount + i]) delete FaceBuffer[FaceCount + i];

            } // Next face

            // Release buffer
            delete []FaceBuffer;

        } // End if FaceBuffer

        return -1;
```

```
    } // End Catch Block
```

With our correctly sized array now allocated and initialized with new 'CFace' object pointers, the next job is of course to release the old 'Faces' array, and overwrite the member variable with the new heap allocated array pointer. Once this is done, we can increment the internal 'FaceCount' variable to ensure that we keep an accurate record of the number of polygons stored in the polygon array.

```
    // Free up old buffer and store new
    if (Faces) delete []Faces;
    Faces = FaceBuffer;

    // Increment face count
    FaceCount += nFaceCount;
```

Finally, with the task of adding new faces complete, this function returns the array element index to the first polygon that has been added. This allows the application to retrieve and iterate through the polygon array beginning with the correct element.

```
    // Return the base face
    return FaceCount - nFaceCount;
}
```

CMesh::BuildFromBSPTree

Although we have not yet discussed our new BSP leaf tree class, we should be familiar enough with the general concepts involved in the construction of leaf trees to know that, once the compilation has completed, the tree will store a list of each of the polygons contained within its leaves. Once the BSP tree has been built, we need to write the polygon information to file in a format that is easy for the application to retrieve. Due to the fact that our application is already capable of loading mesh and polygon data from the IWF file, it makes sense that we export the resulting BSP tree polygon data out to file as a standard mesh. This will prevent us from having to implement another series of polygon import functionality and will allow the BSP tree's polygon data to be loaded seamlessly by *any* IWF import procedure.

This 'BuildFromBSPTree' function has been provided in order to help us convert the BSP tree polygon data into a format compatible with the pre-defined IWF mesh / polygon structures. It can also be used by any additional compilation process in situations where the polygon information must be extracted. We will discuss the process by which the remainder of the BSP tree information is accessed and exported at a later stage, but for now this is a good example of how a mesh might be constructed using existing polygon information.

```
bool CMesh::BuildFromBSPTree( const CBSPTree * pTree, bool Reset /* = false */ )
{
    ULONG Counter = 0, i;

    // Validate Data
    if (!pTree) return false;
```

There are two parameters declared by this function. The first is the BSP tree class instance from which we would like to extract the polygon information. We need not concern ourselves with how this class is defined at this point because our primary focus is on how the mesh is being constructed.

The second 'Reset' parameter allows the application to specify whether or not the polygons contained within the specified BSP tree should be released before the specified BSP tree is processed. This parameter is optional, and has a default value of 'false'. If the application specifies 'true' to this parameter, then the mesh will be reset to a default state before extracting the polygons from the BSP tree and storing them in a newly built polygon array. Specifying 'false' will *prevent* the mesh from being reset. This will result in each of the BSP tree polygons being appended to the already existing mesh polygon array.

The very first task is to test the value of the 'Reset' parameter and reset our mesh to a default state if it found to contain a value of 'true'. This is achieved simply by making a call to the mesh's 'ReleaseFaces' function that is responsible for releasing all of the memory associated with the polygon array and resetting each of the associated member variables to their initial state. This function will be covered shortly. We must also reset the matrix stored within the mesh to an identity state.

```
    // Reset if requested
    if (Reset) { ReleaseFaces(); Matrix.Identity(); }
```

With the mesh having been reset or otherwise left intact, we next begin the task of building the mesh data. Before we do so however we need to determine how many polygons must be allocated in the mesh for the storage of those BSP tree polygons that we are interested in. Although we could simply allocate enough storage space for every polygon within the BSP tree and copy the data regardless of its status, there may be cases in which a polygon in the tree has been marked as deleted but has not physically been removed from the array. In addition, there may be polygons stored in the BSP tree that are invalid, however unlikely this may be. For this reason, it is advisable for this function to validate each of the polygons contained with the BSP tree and count the number of polygons that we will ultimately store.

This is demonstrated in the following code snippet in which we make use of two of the access functions exposed by the 'CBSPTree' class. These functions are named 'GetFaceCount' and 'GetFace' respectively. The 'GetFaceCount' function is responsible for retrieving the number of polygons / faces currently stored within the BSP tree class instance. Likewise, the 'GetFace' function is used to retrieve a pointer to a polygon situated at the specified index in the BSP tree's internal polygon array. This polygon is of a type derived from the 'CFace' class named 'CBSPFace' that is intended to extend our renderable polygon with additional BSP specific information. We will cover this additional polygon class in further detail a little later, but for now it should be fairly obvious as to the purpose of each of these additional variables.

The following code implements this validation technique. Put simply, this code loops through each of the polygons contained within the BSP tree and validates two of its exposed parameters. The first is the 'Deleted' member. In certain situations, such as during the hidden and exterior surface removal processes, polygons may need to be removed. Rather than physically remove the polygons from the tree, they are flagged as deleted. This allows us to undo a split operation if necessary, as we shall see in our coverage of HSR split repair later in this chapter. The second test we perform is simply to validate that this particular polygon contains at least the 3 vertices required to constitute a single triangle. If both of

these tests are passed, then a local counter variable is incremented. Once this loop has completed, the counter variable will contain the total number of polygons that we need to allocate in order to store every polygon that passed the validation tests.

With this tallying process completed, we then attempt to perform the actual allocation of the mesh's internal polygon array via a call to 'AddFaces'. Recall that the 'AddFaces' function returns either '-1' if a failure occurred, or the index to the first polygon added to the internal array should the allocation be successful. We will need this information a little later, so the return value from this function is retrieved and stored. Since the value currently stored in the counter variable we had previously used is no longer required, we can reuse this variable and overwrite its value with that returned by 'AddFaces'.

```
    // Count valid BSP tree faces
    for ( i = 0; i < pTree->GetFaceCount(); i++ )
    {
        if ( !pTree->GetFace(i)->Deleted &&
              pTree->GetFace(i)->VertexCount >= 3 ) Counter++;

    } // Next BSP Face

    // Allocate our faces
    Counter = AddFaces( Counter );
    if ( Counter < 0 ) return false;
```

At this point we have allocated the memory required to store the information for each of the BSP tree polygons that passed the initial validation tests. As a result we are now able to begin copying this polygon data into the mesh's internal array. We begin by looping through each of the BSP tree polygons once again, skipping over any of those which are considered to be invalid using the same two tests as were used in the previous loop.

```
    // Now we loop through and copy over the data
    for ( i = 0; i < pTree->GetFaceCount(); i++ )
    {
        CBSPFace * pTreeFace = pTree->GetFace(i);

        // Validate tree face
        if ( pTreeFace->Deleted ) continue;
        if ( pTreeFace->VertexCount < 3 ) continue;
```

With the validation tests carried out, we are guaranteed to have access to a valid BSP tree polygon at this point. Because of the fact that the BSP tree's 'CBSPFace' class is derived from the 'CFace' class, it might seem that we could simply copy over the pointer to this object without any adverse affects. However, we should not make any assumptions about the ownership of the BSP tree polygon data because there is no way for us to know if the application has further use for this information. If we were to simply copy the BSP tree polygon pointers into the mesh, and were to subsequently release that mesh, the BSP tree's internal array would now contain pointers to an invalid memory location. It is much safer and cleaner for us to perform a 'deep' copy at this point.

The following code does just this. Each of the 'CFace' members are populated using their counterparts in the 'CBSPFace' class retrieved from the BSP tree. Likewise, we also allocated a new vertex array for this new mesh polygon and duplicate the vertex data.

```
        // Retrieve faces for processing
        CFace    * pFace     = Faces[Counter];

        // Copy over data
        pFace->MaterialIndex = pTreeFace->MaterialIndex;
        pFace->TextureIndex  = pTreeFace->TextureIndex;
        pFace->ShaderIndex   = pTreeFace->ShaderIndex;
        pFace->Flags         = pTreeFace->Flags;
        pFace->SrcBlendMode  = pTreeFace->SrcBlendMode;
        pFace->DestBlendMode = pTreeFace->DestBlendMode;
        pFace->Normal        = pTreeFace->Normal;

        // Copy over vertices
        if ( pFace->AddVertices( pTreeFace->VertexCount ) < 0 ) return false;
        memcpy( pFace->Vertices, pTreeFace->Vertices,
                pFace->VertexCount * sizeof(CVertex) );

        // We used a new face
        Counter++;

    } // Next BSP Face
```

Note in the previous code that we used the 'Counter' variable to index into the mesh's newly allocated internal polygon array. This polygon would be the destination object into which this particular BSP polygon data would be copied. Recall that this variable was used to store the value returned from the 'AddFaces' function which specified the first polygon in the newly allocated set. Each time we find a valid polygon within the BSP tree and subsequently store it within the mesh, this counter variable must be incremented in order to move on to the next destination polygon.

The final task in this function, before returning a success code, is to update the mesh's bounding box information to take into account any new polygons that have been added to the mesh. This is achieved with a call to the 'CalculateBoundingBox' member function.

```
    // Calculate our bounding box
    CalculateBoundingBox();

    // Success
    return true;
}
```

Although it may not be clear at this point where and when we might make use of this function, it remains a useful discussion because it allowed us to examine a common usage of the mesh class. As mentioned earlier, we will be utilizing this function later on in the development of our compilation tool at which point any remaining questions you may have about this functions use should be resolved.

CMesh::CalculateBoundingBox

By now you should be very familiar with the techniques involved in calculating a bounding box using mesh polygon and vertex information. This is something we have had to implement many times, in several different parts of our previous run-time applications. Thankfully, the new 'CBounds3' utility class that we have implemented in this lab project performs the majority of the laborious work on our behalf.

The 'CalculateBoundingBox' function accepts no parameters. It simply iterates through each of the polygons contained within the mesh and passes the vertex data into the 'CalculateFromPolygon' bounding box member function. This function accepts a pointer to a set of vertex data, the number of vertices in that set, the size of the vertex structure used – which enables it to step to the positional information of each vertex in the array – and a final 'Reset' parameter that dictates whether to calculate new bounding box extent vectors from only that polygon, or to grow those extent values already contained within the object. In the implementation of this function we need to grow the bounding box for each polygon that is contained within the mesh. To this end, we reset the bounding box in advance and then specify 'false' to the reset parameter of the bounding box's member function.

Finally, once the mesh's 'Bounds' member has been calculated, this function returns to the calling function with a reference to this variable. This allows the application to use the result of the operation performed by this function as the input to another, without having to access the bounds member separately. Because this is achieved via the function return value, this is of course optional and the calling function need not consider this result.

```
const CBounds3& CMesh::CalculateBoundingBox()
{
    // Reset bounding box
    Bounds.Reset();

    // Loop through each face
    for ( long i = 0; i < (signed)FaceCount; i++ )
    {
        Bounds.CalculateFromPolygon( Faces[i]->Vertices, Faces[i]->VertexCount,
                                     sizeof(CVertex), false );

    } // Next face

    // Return our bounds reference
    return Bounds;
}
```

CMesh::ReleaseFaces / ~CMesh()

The final two functions in this class are those responsible for cleaning up and releasing any resources allocated by the mesh object. The first is a public method named 'ReleaseFaces'. This function is responsible for releasing only that memory allocated for the internal polygon array in addition to each of the heap allocated polygon objects stored within that array. As always, each value is tested to see whether it contains a value other than NULL. If this found to be the case then that object or array can be safely released. While this function can be called by the application at any point if it wishes to reuse a particular mesh object, this function is also called by the second of these two functions – the class

destructor – as the object is released or goes out of scope. In addition to calling the 'ReleaseFaces' function, the class destructor also releases any memory allocated for the storage of the 'Name' member string.

```
void CMesh::ReleaseFaces()
{
    // Clean up after ourselves
    if ( Faces )
    {
        // Release all face pointers
        for ( long i = 0; i < (signed)FaceCount; i++ )
        {
            if ( Faces[i] ) delete Faces[i];

        } // Next face

        // Release the array itself
        delete []Faces;
        Faces = NULL;

    } // End if Faces

    FaceCount = 0;
}
```

```
CMesh::~CMesh()
{
    // Clean up after ourselves
    ReleaseFaces();
    if ( Name ) delete[] Name;
}
```

Prerequisite Class Conclusion

At this stage we have now covered each of the prerequisite classes and functionality that we will need to have at our disposal as we move forward with the development of our new compilation tool. Although it may seem as though we have been covering a lot of topics that we have already discussed in previous chapters, there are many nuances to this new set of utility classes that we had to be aware of both in their implementation and their usage. Even though we have not yet begun to discuss the core implementation of the new pre-processing application, we have built a solid framework on which we can build that will make the job of creating both the tool itself, and each of the compiler modules a much more simple task.

With this new arsenal of utility classes at our disposal, let us now move on to the discussion of the application core.

Application Front End – Main.cpp & LogOutput.cpp

We discussed previously that we wanted to build a modular and portable tool that would enable us to integrate the core compiler functionality with any front end application that we might design. This could be a standalone GUI tool that serves only as the container for the compiler itself, or it could be another more comprehensive tool such as a level editor or modelling package. In lab project 16.2, we will be constructing a simple console like application that will serve as the front end for the compiler functionality. This will ensure that the code required to provide the interface does not obfuscate the more important compiler logic.

If we open and examine the source modules contained in the 'Application Source' folder of our source project, it is clear that the actual wrapper application that we will construct really is nothing more than the logic required to open a console window and output progress and log information to the end user. Given that the logging procedures constitute such a large portion of this lab project's responsibilities, let us quickly summarize how this process works.

Earlier in this chapter we touched briefly upon the mechanism by which the compiler integrates itself into the application for the purposes of writing progress and status information. Recall that this is achieved by using a base interface from which the application must derive. This base interface is named 'ILogger' and is declared in the header file 'Common.h' which can be found in the 'Support Source' project directory. This pure abstract interface class declares several virtual functions which must be implemented by the application derived class.

The following list outlines the purpose of each of these interface functions.

```
void LogWrite( unsigned long Channel, unsigned long Flags, bool NewMessage,
               LPCTSTR Format, ... );
```

This function is intended to be called by any part of the application to output messages to the user.

As a high level concept, the logging class is required to keep track of the information being output by several sources. These are referred to as channels within this implementation. Put simply, the application may be required to output the information for various different compiler process tasks that may be executed. These would be in addition to the general process status information that our application may need to display. Each compilation process is assigned a unique channel index that only that specific process will ever pass to the logging class member functions. The first parameter to the 'LogWrite' interface function is this channel index used to signify the destination to which this message is intended.

The second parameter can be passed one or more of the following flags:

```
#define LOGF_WARNING     1      // Log Flags : Display with warning format
#define LOGF_ERROR       2      // Log Flags : Display with error format
#define LOGF_UNDERLINE   4      // Log Flags : Display with underline property
#define LOGF_BOLD        8      // Log Flags : Display with bold property
#define LOGF_ITALIC      16     // Log Flags : Display with italic property
```

The derived class is not required to obey these flags due to the fact that in certain circumstances, such as in a console application, not all of these flags are even able to be supported. As a result, when calling the 'LogWrite' function, the values we specify should be considered to be only hints.

The third parameter, 'NewMessage' is used to inform the logging class whether or not this is the start of a new message. If a value of false is passed to this parameter then the logging class should consider this message to be a continuation of the previous.

The fourth and final defined parameter is the 'Format' value. You may have noticed that this method is declared using an ellipsis modifier. This format parameter is intended to be used in the same way as we would use the format parameter in the 'printf' function. If you are developing a derived logging class of your own, it is recommended that you look up the 'va_start', 'va_arg', 'va_end' and 'vsprintf' C++ runtime functions that can be used to interpret the format value in conjunction with variable argument lists.

```
void SetRewindMarker( unsigned long Channel );
void Rewind         ( unsigned long Channel );
```

This function is intended to cause the logging class to store a value indicating the current position within the specified channel when this function is called. This is used in conjunction with the 'Rewind' function to have the logging class roll back the channel text to the marker location recorded earlier. This is incredibly useful in tool environments like this where we might want to constantly update a piece of text within the log text with a percentage progress counter for instance.

```
void Clear( unsigned long Channel );
```

The derived class should clear the specified log channel of all text and marker information whenever this function is called.

```
ULONG GetCurrentChannel( );
```

Whenever a call is made to the 'LogWrite' function, the derived log class should record the most recently specified channel index. This value can then be retrieved by the application to continue writing to the same log channel as it wrote some initial text for instance.

```
void SetProgressRange( long Maximum );
void SetProgressValue( long Value );
```

The 'SetProgressRange' function is the means by which the application informs the derived logging class of the maximum value it might pass to the 'SetProgressValue' function. As an example, we might specify a value to the 'SetProgressRange' function of 900. With this value as a maximum, when we then pass a value of 450 to the 'SetProgressValue' function, the logging class would interpret this value to mean '50%' (i.e. 450/900 = 0.5 * 100 = 50).

```
void UpdateProgress( long Amount = 1 );
```

The 'UpdateProgress' function is the means by which the application has the logging class print the current progress percentage value to the current logging channel. Optionally, this function can be passed

a value used to increment the progress value as if it was passed in to the 'SetProgressValue' function. As an example, we might have set our maximum progress range to be equal to the number of polygons that we want to process. We might then call the UpdateProgress function, passing an amount value of '1', each time we processed another polygon. In this case the logging class would automatically output a value of '100%' at the moment we processed the final polygon and called 'UpdateProgess' for the final time.

```
void ProgressSuccess( unsigned long Channel );
void ProgressFailure( unsigned long Channel );
```

These two functions are utility functions that can be called to have a success or failure message output to the specified channel in a format decided by the derived logging class developer. This might also allow the logging class to take further action when a success or failure occurred such as sending an e-mail to a technician should any process fail.

We will not go into any further detail about how we have implemented the derived console output class in this workbook. If you would like to take a look at the source for this class you can do so in the 'LogOutput.cpp' file in the 'Application Source' project directory.

With these general application concepts out of the way, lets move straight on to the application entry point function.

WinMain

Although we are constructing a console *like* application in this lab project, we have used a standard Win32 project to enable us to gain access to all of the benefits provided by such an application. This includes the ability to create and use the Windows™ common dialogs such as the standard 'Open' and 'Save' dialogs. In addition to supporting common Win32 API functionality, building an application of this type allows for future Win32-centric enhancements an example of which might be the inclusion of an additional window in order to preview the compiled scene perhaps.

Because we are developing our front end as a Win32 application, our entry point is defined using the standard 'WinMain' function, as opposed to the 'main' function we would use if we were creating a native console application. You should already be familiar with the WinMain entry point function at this stage, so we will not go into any further detail about the parameters or their intended use.

```
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR lpCmdLine,
                    int iCmdShow )
{
    OPENFILENAME  File;
    TCHAR         FileName[MAX_PATH];
    CLogOutput    LogOutput;
    CCompiler     Compiler;
    bool          bSuccess;
    ULONG         i;
```

Notice in the above code that we have defined several local variables for use within this function. By far the most important of these is the 'Compiler' variable of the type 'CCompiler'. This class is the one that

provides our application with its entire scene processing ability. We will be discussing this class in the next section of this chapter, but for now let us examine how the application might make use of the functionality exposed by this compiler class.

Due to the fact that this is not a native console application, the very first thing that our application must do is to open a console window that will serve as the channel through which the user is informed about the progress and status of each of the compiler tasks to be executed. This is achieved via a call to the Win32 'AllocConsole' function. This function will create the console window for use by our application but will *not* redirect the standard output to this window automatically. Functions and template classes such as 'printf' and 'cout' are designed to output string data through the 'stdout' data stream also know as the standard output stream or channel. In a native console application this data stream is automatically attached to the console that is implicitly created when the application begins. This would result in any string data passed to a call to 'printf' being output to the console window as we would expect. In a Win32 application however, this is obviously not the case because no console window exists without our explicit instruction. Therefore, we must also perform this operation ourselves by attaching the 'stdout' stream to the newly created console window once it has been created.

In C++, the standard output stream is defined by a global variable most easily accessed using the name 'stdout'. This is actually a variable of the type 'FILE' which is a structure provided by the standard C runtime library in the 'stdio.h' header file. You should already be familiar with this structure because a pointer of this type is returned and passed to functions we have used extensively in the past such as 'fopen' and 'fread'. Despite the name of this structure, the 'FILE' type can apply to any sort of data stream be that a literal file, or other data buffers such as that maintained by a simple console window. So, given the fact that 'printf' writes the string data out to the 'stdout' *file* we can take advantage of this fact by making a call to the 'freopen' function that allows us to redirect the output of an already existing data stream – such as 'stdout' – to a new destination. This function works in a similar fashion to the 'fopen' function in that it accepts a filename and a mode used to describe how the stream should be accessed. However, it also accepts a pointer to an existing 'FILE' object that we would like to redirect.

Obviously we do not want to save the data written to the standard output stream to a file and yet the first parameter of this function is expecting a filename. So how exactly does this function help us place the output of calls such as 'printf' to a console window? We can in fact utilize a feature of the underlying operating system to enable us to output string data to the console by specifying a string of "CONOUT$" to the path parameter of this function. This is essentially a special path name that is reserved for just this type of situation. In fact, we can use this path name with many of the file handling functions in order to access and output data to and from the console window. In order to integrate the standard output functions with the console window we must also pass "a" (or append) to the mode parameter, and finally the 'stdout' stream pointer variable to the final parameter.

With the 'stdout' *file* stream now redirected to the new console window, we should find that any function or library designed to write information to the standard output stream will now seamlessly output to our application's console window.

```
    // Create a console window so that we can use it as our logging output
    AllocConsole();

    // Redirect all standard output to console (i.e. printf, cout etc).
```

```
    freopen("CONOUT$", "a", stdout);
```

Now that the main window into our application has been created and attached, we can now begin to output logging and progress information to the user. The following code first initializes the instance to our custom 'CLogOutput' class by calling the 'Create' function. We pass the total number of channels that we would like it to maintain to this function. We use an arbitrary figure of 20 in this code to ensure that there are enough to be going on with.

Once we have initialized the logging class, we then output the leading messages for this application such as its name, version and copyright information. Refer back to our discussion of the base interface 'LogWrite' function in order to understand the meanings of the specified parameters.

Finally, we pass a pointer to the instance of our console logging CLogOutput class into the compiler. This will then be distributed throughout the system to allow the various components to output progress and status information to the user.

```
    // Create the log output handler and attach it to the compiler
    LogOutput.Create( 20 );
    LogOutput.LogWrite( LOG_GENERAL, 0, false,
                        _T("\nSolid Leaf BSP Tree Compiler v1.0.0\n"));
    LogOutput.LogWrite( LOG_GENERAL, 0, false,
                        _T("Copyright © 2005 GameInstitute.com.")
                        _T("All Rights Reserved.\n"));
    LogOutput.LogWrite( LOG_GENERAL, 0, true,
                        _T("Compiler startup successful, awaiting user input."));
    Compiler.SetLogger( &LogOutput );
```

The only piece of information that we need from the user in this application is the filename of the IWF file that they wish to be imported and passed through the various compilation procedures. In this application, rather than using a potentially cumbersome command line parameter scheme, this is achieved by using the Win32 'GetOpenFileName' function and associated structures that have been used and discussed in previous lessons. If the user selected a valid file then the 'GetOpenFileName' function will return a value of true, after storing the selected path and filename information in the variable specified through the 'lpstrFile' member of the 'OPENFILENAME' structure passed to this function. In our implementation this would be the local string variable named 'FileName'. If the user chose to cancel the open dialog, or some other failure occurred, then this function will return false and our application will exit.

```
    // Fill out our default file dialog structure
    ZeroMemory( &File, sizeof(OPENFILENAME) );
    ZeroMemory( FileName, MAX_PATH * sizeof(TCHAR));
    File.lStructSize  = sizeof(OPENFILENAME);
    File.hwndOwner    = NULL;
    File.nFilterIndex = 1;
    File.lpstrTitle   = _T("Select Input IWF File for Compilation.");
    File.Flags        = OFN_EXPLORER | OFN_HIDEREADONLY | OFN_FILEMUSTEXIST |
                        OFN_NOCHANGEDIR | OFN_PATHMUSTEXIST;
    File.lpstrFile    = FileName;
    File.nMaxFile     = MAX_PATH - 1;
```

```
    File.lpstrFilter  = _T("All Supported Formats\0")
                        _T("*.IWF;*.BWF\0")
                        _T("Interchangable World Format (*.iwf)\0")
                        _T("*.IWF\0");

    // Retrieve a filename.
    if ( !GetOpenFileName( &File ) ) return 0;
```

Once the user has selected the file that they would like to be processed, the application can then inform the compiler class of the user's selection via a call to the 'SetFile' function and begin the compilation process with a call to 'CompileScene'. It is important that we store the return value from the latter of these two functions because it will be used later to determine if the compilation process was successful or not.

```
    // Inform the compiler about the file to compile
    Compiler.SetFile( FileName );

    // Compile the scene
    bSuccess = Compiler.CompileScene();
```

In this application, these two lines of code constitute the majority of the application's interaction with the core of our compiler. Although it is possible to play a much more active role in deciding which processes the user would like to perform on the scene geometry, in addition to setting several available options for each processor module, we will use the default settings in this application. These default settings specify that every available compiler task should be enabled and executed using sensible process parameters. We will see the means by which we can construct and specify these options in the next section when we cover the actual compiler classes shortly.

With the compilation process completed, our application then proceeds to clear the console window using one of the handy member functions exposed by our console logging class – 'ClearConsole' -- and then requests that all of the logged process information for each channel is output to the console as a last step. This is achieved via a call to the 'CLogOutput::SwitchChannel' channel function. This allows the user to see the final results of each task undertaken by the compiler. If a failure was encountered, the user will be able to identify the step at which this failure occurred and potentially remedy that situation.

```
    // Clear the console window and output all of the status text one last time
    LogOutput.ClearConsole();
    for ( i = 0; i < 20; ++i )
    {
        // Was there any text in this channel?
        if (LogOutput.GetChannelText( i ) != NULL)
        {
            // Print the channel information out
            LogOutput.SwitchChannel( i, false );
            printf( _T("\n\n-------------------------------------------\n" );

        } // End if anything in channel

    } // Next Channel
```

If the compilation process was a complete success then the return code we previously stored in the local Boolean variable ‘bSuccess’ will contain a value of true at this stage. If this is found to be the case then we can instruct the compiler to save the resulting data to a new file as specified by the user. The next task therefore is to request that the user input a location and filename into which the compiled scene information will be exported.

Because we had previously initialized the ‘OPENFILENAME’ structure that was passed into the earlier call to the Win32 open dialog function, we need only alter those structure variables that must change in order to display the common *save* dialog with a call to the Win32 ‘GetSaveFileName’ function. If the user selected a valid output path and file name within this dialog, then the ‘GetSaveFileName’ function will return true, after having copied the selected file and path name into the local variable ‘FileName’ as before. This file name string can then be passed into the compiler object’s ‘SaveScene’ method in order to have the newly compiled scene data exported to that location.

If the compiler failed for any reason, or the user chose to cancel the save dialog, then the scene export request will be bypassed altogether as we move onto the closing stages of this function and our application front end.

```
    // Save the scene if it compiled succesfully
    if ( bSuccess )
    {
        // Alter structure for any changed options
        File.nFilterIndex = 1;
        File.lpstrTitle   = _T("Select Output BWF File.");
        File.lpstrFilter  = _T("BSP Compiled IWF (*.bwf)\0*.BWF\0");
        ZeroMemory( FileName, MAX_PATH * sizeof(TCHAR));

        // Retrieve a filename.
        if ( GetSaveFileName( &File ) )
        {
            // Save the scene
            Compiler.SaveScene( FileName );

        } // End if Selected File
        else
        {
            // Output information
            printf( "User chose to cancel save operation." );

        } // End if no file

    } // End if success
```

One of the downsides of running a console application in a windows environment is that as soon as the application exits, the console window is automatically closed and destroyed. It is often the case however, that the application may have output several pieces of important information to the console that the user may not see if the window were to close as soon as the process was complete. For this reason we include a call to the runtime ‘_getch’ console function which is designed to retrieve character input from the keyboard in a synchronous fashion. This means that the ‘_getch’ function will cause our application to wait until a key has been pressed before program control is returned and the remainder of

the code is executed. When combined with a standard 'Press any key' message, this will allow the user to examine any messages output to the console window at their leisure.

```
    // Hold application until user presses key
    printf( "\n\nPress any key to exit..." );
    _getch();
```

The final piece of code within this function simply releases the console window that we allocated earlier, and returns from the WinMain function with an exit code of 0. This will signal the end of the application's lifetime and will then exit.

```
    // Clean up console
    FreeConsole();

    // Return
    return 0;
}
```

With our application front end fully implemented, let us now focus on the inner workings of our pre-processing application by examining the core compiler classes.

The Compiler Core – CCompiler.cpp

The core of the compiler tool we are developing in this lesson is centered on the 'CCollision' class found in the 'CCompiler.cpp' source file. This class is the primary wrapper around the various tasks that our pre-processing compiler tool must undertake. Rather than require that the application be heavily involved in the compilation process, this class provides a simple interface through which much of the common functionality required by the pre-processing tool can be executed without any further involvement by the application. This common functionality includes:

1) Loading all of the required scene information from file. This information will include the scene data that is to be passed into the various compiler processes, as well as the scene data that must be exported back out to the resulting file.

2) Setting the options for each compiler task that has been selected for execution by the application.

3) Performing any high level logic for setting up the data required as input into each compilation process, and also to interpret any pieces of resulting information that may need to be merged back into the scene data set managed by the compiler class.

4) Managing and executing any compiler process modules that have been enabled by the application.

5) Writing the compiled information back out to a new file.

We have made several references in the past to individual compiler modules. Recall that one of the development goals of this tool was to create a modular core that could be easily extended to include

additional pre-processing tasks in the future. With this design concept in mind, each of the individual compilation / processing tasks that we will be implementing in this tool are separated into individual classes that are each responsible for performing only a smaller portion of the overall scene compile process. The following list outlines some of the modular processing classes that we will encounter. The first three in this list are covered in this lesson, and the final two will be covered in the next.

CProcessHSR
This procedure in effect pre-processes our scene geometry and removes any 'Illegal Geometry' ready for processing by the BSP compiler. These fragments of illegal geometry are described as ‘hidden surfaces’ due to the fact that any surfaces – or fragments of surfaces – that intersect the solid area of another part of the scene cannot usually be seen due to depth-based pixel culling. These fragments are potentially destructive and often cause complete failure when attempting to generate solid and empty leaf information within the BSP compiler. For this reason, these illegal polygon fragments must be removed.

This process works at the mesh level, meaning the scene data must be in its primitive form on import. Once processing is complete, all illegal geometry should have been removed, and the mesh data merged into a single set of polygons ready for BSP compilation.

CBSPTree
The leaf-based BSP compile process is used, among other things, for partitioning the scene (along each polygons plane) and collecting a series of convex polygon clumps, or leaves, that make up the scene. These 'nodes' and 'leaves' are then utilised by many procedures we may wish to perform on our scene database, such as portal and PVS compilation, constructive solid geometry (including HSR) and much more. The tree itself can then also be used at runtime for many operations such as collision detection and line of sight.

CProcessTJR
T-Junctions are often (not exclusively) the result of a surface being split, while its neighbour remains intact. This has the effect of splitting the edge of one surface, adding at least one vertex, while the edge of the adjacent polygon remains in tact. The problem with this situation is that it often results in artifacts during rasterization due to the differing amounts of floating point errors that occur while traversing each of these edges. These artifacts are often called 'Sparklies' or 'Sub-Pixel Gaps' where certain pixels along those edges are not rendered. This allows whatever color pixels may have been rendered behind to show through. This is not the only problem caused by T-Junctions however. Because the vertices of these two surfaces no longer link exactly, moving the centre vertex along one edge will not adjust the adjacent edge in the same manner. This process locates such problematic edges, and attempts to repair the problem by inserting a vertex on the adjacent edge.

CProcessPRT
In order to calculate the visibility between our scene's leaves, we need to determine the shape and location of the gaps that exist between them. Once we have this information, we can then proceed to use it in order to determine what is visible through the gaps these portal polygons now occupy.

Essentially this procedure feeds extremely large initial surfaces through the tree at each node, clipping it to every other subsequent node, as it traverses the tree. This process continues until the fragments of such surfaces reach an empty leaf. These remaining fragments, assuming they pass final validation,

accurately describe these gaps and become portals between our scene's leaves ready for visibility processing among other things. **Note: This process is not discussed until the next lesson.**

CProcessPVS
The PVS compilation procedure ultimately determines the 'potential' visibility between the leaves within our scene. For every leaf contained within the BSP tree, this process will determine which other leaves in the scene are visible from any position within that particular leaf. This is an extremely intensive process and can take several orders of magnitude longer to complete than every other compile process combined. **Note: This process is not discussed until the next lesson.**

The CCompiler Class

In this lab project we will only need one instance of a 'CCompiler' object that will be created and managed by the application front end discussed in the previous section. Although this application will only ever use one instance of this type, the compiler class has been designed to provide an independent, standalone interface that allows the application to easily manage multiple compiler class instances. This would allow the application to use multi-threading in order to maintain more than one compiler process that can be run concurrently – compiling multiple levels at any one time – or to simply allow the compiler to run in the background whilst the user carries on working with the tool / application into which the compiler core has been integrated.

Before we move on to the implementation of the compiler class functionality, let us begin by examining the declaration for the main compiler class. We have removed the list of member functions to improve readability, so check the source code for a full list of methods. All we are interested in at the moment are the member variables and structures declared within the class scope.

```
class CCompiler
{
public:

    // Constructors & Destructors for This Class.
             CCompiler();
    virtual ~CCompiler();

    // Public Variables for This Class.
    vectorMesh      m_vpMeshList;       // A list of all meshes loaded.
    vectorTexture   m_vpTextureList;    // A list of all textures loaded.
    vectorMaterial  m_vpMaterialList;   // A list of all materials loaded.
    vectorEntity    m_vpEntityList;     // A list of all entities loaded.
    vectorShader    m_vpShaderList;     // A list of all shaders loaded.

    // Public Member Functions Omitted

private:

    // Private Variables for This Class.
    HSROPTIONS      m_OptionsHSR;       // Hidden Surface Removal Options
    BSPOPTIONS      m_OptionsBSP;       // BSP Compilation Options
    TJROPTIONS      m_OptionsTJR;       // T-Junction Repair Options
```

```
    ILogger         *m_pLogger;          // Interface used to log progress etc.
    LPTSTR           m_strFileName;      // The file used for compilation
    COMPILESTATUS    m_Status;           // The current status of the compile run
    ULONG            m_CurrentLog;       // Current logging channel for messages.

    CBSPTree        *m_pBSPTree;         // Our compiled BSP Tree.

    // Private Member Functions Omitted

};
```

There are several important member variables declared within this class that we must first examine, so let us briefly discuss each of them.

vectorMesh m_vpMeshList
This member variable will be used to store a list of all of the meshes loaded from the source IWF scene file. It is of type 'vectorMesh', which is a typedef for an STL vector of 'CMesh' pointers.

```
typedef std::vector<CMesh*>       vectorMesh;
```

This STL vector is populated by the IWF loading code covered later in this lesson. Each mesh object stored within this array will potentially be processed by the compiler as each compilation module is executed.

vectorTexture m_vpTextureList
vectorMaterial m_vpMaterialList
vectorEntity m_vpEntityList
vectorShader m_vpShaderList
These four member variables serve as the means by which our compiler stores the static scene information that will not be processed by the compiler portion of the application. Instead this information is only maintained in order for it to be exported back out to the destination file once the scene has been processed. The type of each of these member variables is a typedef of one of the STL vector items outlined in the following list.

```
typedef std::vector<iwfTexture*>  vectorTexture;
typedef std::vector<iwfMaterial*> vectorMaterial;
typedef std::vector<iwfEntity*>   vectorEntity;
typedef std::vector<iwfShader*>   vectorShader;
```

Due to the fact that we are simply maintaining this information so that the final scene can be exported correctly, each of these STL vector items is simply designed to store pointers to each type of storage class provided directly by the IWF import library. As with the mesh information, these STL vectors are populated by the IWF loading code covered later in this lesson.

HSROPTIONS m_OptionsHSR
Each compiler module in this application has its own unique options structure that can be used by the application to alter the way in which each module performs its task – if it is to be executed at all. These structures can be found in the 'CompilerTypes.h' header file in the 'Compiler Source' project directory.

Although we do not alter these option values from their default state in this lab project, an application can access and modify the information stored in these options structure members by calling the appropriate 'GetOptions' or 'SetOptions' functions defined by the 'CCompiler' class.

This member variable contains those options available to the application for the hidden surface removal process implemented within this lab project.

```
typedef struct _HSROPTIONS
{
    bool Enabled;

} HSROPTIONS;
```

We can see that the 'HSROPTIONS' structure contains only a single Boolean variable named 'Enabled'. This variable can be used by the application to specify whether or not the scene data should be passed through the hidden surface removal process. By placing a value of 'true' into this variable we are instructing the compiler that we would like to enable the HSR process in order to ensure that all of the meshes are unioned together into a single valid mesh ready for BSP compilation.

BSPOPTIONS m_OptionsBSP

In a similar fashion to the hidden surface removal module, there are also a set of options available that allow us to alter how the binary space partitioning module will compile the final BSP leaf tree. This member variable stores these BSP module options and is declared with the type 'BSPOPTIONS'. This structure is outlined below.

```
typedef struct _BSPOPTIONS
{
    bool            Enabled;
    unsigned long   TreeType;
    float           SplitHeuristic;
    unsigned long   SplitterSample;
    bool            RemoveBackLeaves;
} BSPOPTIONS;
```

In addition to the 'Enabled' flag – used to instruct the compiler as to whether or not this process should be performed – the 'BSPOPTIONS' structure defines several options that dictate how the final scene BSP tree will be compiled. The purpose of each of these option values are detailed in the following list.

- *TreeType*
  In the previous spatial hierarchy compilation classes that we have implemented, we exposed the ability to compile a tree that did not split the polygon data against the separating planes as the tree was being constructed. While compiling the tree without splitting polygon data was not suitable for the hardware rendering techniques we constructed to handle the tree data, there were many non-rendering situations in which storing unsplit polygon data in the leaves of the tree may have been beneficial to us.

  If you have read the textbook for this chapter, you should know that splitting the polygon data during the compilation of a polygon aligned BSP leaf tree is critical and cannot be avoided. This

is due to the fact that the splitting of the source polygon data is a key aspect of how the BSP tree forms its convex leaf areas in addition to providing valid solid and empty leaf information. Although the splitting of polygons cannot be avoided during the compilation of the BSP tree itself, it is possible to adapt the way that polygon data is stored in the leaves in order to achieve a similar effect. We will not go into detail about how this process works at this point and we will defer this discussion until the coverage of our new BSP tree compilation class. Just know for now that the BSP tree compiler will have the ability to generate a tree which can store unsplit polygon information.

This option variable is intended therefore to instruct the BSP compiler as to whether or not it should generate the final scene tree using this non-split concept. This tree type variable should be set to one of the two defined constants shown below.

```
#define BSP_TYPE_NONSPLIT   0
#define BSP_TYPE_SPLIT      1
```

The first of these two constants instructs the compiler to generate a tree that stores unsplit polygon data, and the second will cause the BSP tree to be generated using the more traditional split polygon approach.

Note: If the BSP tree is intended for use by a rendering engine, then it is recommended that you select the 'BSP_TYPE_SPLIT' option. Because the non-split solution will result in certain polygons ending up in multiple leaves, selecting the traditional splitting option will prevent duplicate polygons from being rendered when using a static vertex / index buffer solution.

- *SplitHeuristic*
  This variable defines the splitting heuristic value that will be passed into the BSP compiler's 'SelectBestSplitter' function. We have already discussed the purpose of this value in detail earlier in this lesson, but to summarise this value allows us to specify the importance of building a balanced BSP tree versus reducing the number of polygons which get split during the compilation process. A good default value to specify for this heuristic variable is one of around 3.0 although you should experiment with this in order to get the best results for the specific requirements of your application.

- *SplitterSample*
  In addition to the SplitHeuristic parameter, the 'SelectBestSplitter' function also accepts a SplitterSample value which can be specified by the application through this option structure member. Recall that this concept allows us to specify the maximum number of potential polygons that will be considered for use as a separating plane by this function. This is simply an early out mechanism that allows us to select a polygon from the first 'n' polygons in the list passed to the selection function rather than testing the entire set. By reducing the number of potential splitters tested during this select step we can greatly reduce the time it takes to compile the polygon aligned BSP tree. This can be useful during the development of our application in which we might only want to build the tree quickly for testing purposes. This member can be set to 0 however when we are compiling a *release* version of the level, in order to have the 'SelectBestSplitter' function test every polygon in the specified polygon list.

- *RemoveBackLeaves*
  When compiling a polygon aligned BSP leaf tree, we have the added advantage of being able to determine the solid and empty areas within the compiled scene. As you should know if you have already been through the textbook for this chapter, the integrity of this solid information is entirely dependant on the validity of the geometry being used. If the scene geometry has been created in a manner compatible with the solid BSP leaf tree, then we should find than no polygons end up behind any terminal node within the tree structure. There are situations in which this is not the case however, even when the input geometry is valid. Due to the inherent inaccuracy of single precision floating point values on today's processors, the possibility still exists that there will be slight errors within either the source polygon data, or those polygons generated during the BSP compilation process. These small, potentially accumulating errors could lead to a situation in which a small fragment of a polygon ends up behind a terminal node that would traditionally be considered solid space.

  This flag allows the application to specify whether or not it would like the BSP compiler to enforce the removal of any polygon fragments which end up behind a node where solid space should exist. Whilst this technique does resolve many situations in which erroneous polygon fragments might fall into solid space, it should not be considered to be a replacement for hidden surface removal.

  Of course, if your application is not interested in maintaining this solid and empty space information, then it is possible to set this variable to a value of 'false'. In this case, a leaf will be generated behind those terminal nodes in which any polygons are found to exist. This is done in a manner similar to that when creating a leaf that will be attached to the front of a node.

  Again, we will look at how this process is implemented later in this chapter during the coverage of our new BSP tree compiler class.

As with each of these compiler option member variables, the 'm_OptionsBSP' member can be retrieved and altered by the application via the 'GetOptions' and 'SetOptions' compiler functions.

TJROPTIONS m_OptionsTJR

The final member variable used to store the options for an individual compiler process is the 'm_OptionsTJR' variable. This variable is of the type 'TJROPTIONS' which describes any settings available to the application for the T-junction repair compiler module. Similar to the 'HSROPTIONS' structure, this contains only a single Boolean variable that instructs the compiler as to whether or not the T-junction repair process should be enabled. In the interest of completeness, this structure is shown below.

```
typedef struct _TJROPTIONS
{
    bool Enabled;

} TJROPTIONS;
```

ILogger * m_pLogger

Earlier we discussed how the compiler portion of the application integrates itself with the front end interface using a base class named 'ILogger'. This interface is used throughout the various compiler modules in order to report any progress, success or failure information to the user. Recall that the 'ILogger' interface is a pure abstract base class that cannot be instantiated in its own right. This forces the application to derive a class from this base interface in which the code required to display the compiler's status information to the user is implemented. In order to maintain the independence of the compiler core from the application front end, this member variable stores a pointer to an instance of the applications log output object using the type 'ILogger' rather than a pointer to any application specific class. This variable should be set by the application with a call to the 'CCompiler::SetLogger' method, passing a pointer to its own logging class instance.

LPTSTR m_strFileName

This member variable stores a string that specifies the path and file name of the IWF file to be imported and processed. This variable is populated by the application using a call to the 'SetFile' method of the compiler class. This variable should be set *before* the compilation process is triggered as this string is read by the compiler when importing the scene geometry that will be subsequently passed through each processing module.

COMPILESTATUS m_Status

Earlier in this section we discussed how the compiler class was designed to function in a multi-threaded environment as either a single background process, or to run concurrently with other compilation tasks. In order to afford the application some control over the execution of each compilation process, this class exposes the ability to both query and modify the running state of the object in order to pause, resume or cancel the currently executing operation. We will discuss how the application can manipulate the state of the compiler shortly.

The 'm_Status' variable stores the current state of this compiler object and can be retrieved by the application with a call to the 'GetCompilerStatus' method of this class. This may contain any one of the four values declared in the following enumerator listing.

```
enum COMPILESTATUS
{
    CS_IDLE         = 0,
    CS_INPROGRESS   = 1,
    CS_PAUSED       = 2,
    CS_CANCELLED    = 3
};
```

Although the meaning of each status item should be relatively obvious given its name, let us just briefly examine these values and the situations in which they may be returned.

- *CS_IDLE*
  This enumerator item specifies that the compiler is currently in an idle state. This means that no compilation process is currently executing. A value of CS_IDLE will only be returned in those cases both before and after the 'CompileScene' function has been executed in order to run the compilation process.

- *CS_INPROGRESS*
  This item specifies that the compiler object is currently in the process of compiling the scene data. As soon as the application calls the 'CompileScene' function, the 'm_Status' variable will be assigned this value to signify that it has begin the operation. If this value is returned, it also means that the compiler is not currently in a paused state and has also not been cancelled by the user.

- *CS_PAUSED*
  As mentioned, the application is able to put the compiler object into a paused state that will temporarily halt any currently executing compilation task. This process can be resumed once again at a later stage should the application choose to do so. We will discuss the method by which this is achieved shortly.

- *CS_CANCELLED*
  In addition to pausing the compilation process, the application also has the ability to cancel the operation all together. Once this cancel request has been received, the 'm_Status' variable will be updated to contain this value. This informs both the various compilation modules, in addition to the application, that the task is currently in the process of being cancelled. As soon as the executing compiler module has been able to gracefully clean up and exit, the status variable will once again be assigned a value of 'CS_IDLE'. During the time in which the status variable contains the 'CS_CANCELLED' value, the application should wait for the compiler to finish before taking any further action with this compiler object.

ULONG m_CurrentLog
This member variable is maintained in order to store the log channel index for the compilation module that is currently executing. When the application's running state is altered, such as having been paused or cancelled, this variable is used to allow the compiler object to output a notification to the log channel for the current task.

CBSPTree * m_pBSPTree
The primary goal of our pre-processing tool is to build a polygon aligned BSP leaf tree based on the information imported from the IWF file specified by the application. Should the application have enabled the BSP compilation process, this member variable will store a pointer to the 'CBSPTree' object that has been compiled from this scene data.

Let us now discuss the functions to the CCompiler class that we have not yet seen. We will start by looking at the initialization functions – such as the constructor, and the member variable accessor functions. We will then examine the methods charged with importing and processing the level data.

CCompiler::CCompiler()

We begin our coverage of the 'CCompiler' methods with the class constructor. This is the function in which each of the compiler class member variables and structures are initialized with a sensible set of default values. In lab project 16.2, the front end interface does not require that the user specify any of the compilation options we discussed earlier. As a result, the compiler and each compilation process will use those default settings specified by this function.

```
CCompiler::CCompiler()
{
    // Set up default HSR options
    m_OptionsHSR.Enabled            = true;

    // Set up default BSP options
    m_OptionsBSP.Enabled            = true;
    m_OptionsBSP.TreeType           = BSP_TYPE_SPLIT;
    m_OptionsBSP.SplitHeuristic     = 3.0f;
    m_OptionsBSP.SplitterSample     = 60;
    m_OptionsBSP.RemoveBackLeaves   = true;

    // Set up default TJR Options
    m_OptionsTJR.Enabled            = true;

    // Reset Vars
    m_strFileName = NULL;
    m_pBSPTree    = NULL;
    m_pLogger     = NULL;
    m_Status      = CS_IDLE;

}
```

We begin by enabling both the hidden surface removal processor and BSP compilation modules. In addition to enabling the latter of these two processes, we also specify the default values for the remainder of the BSP compiler settings.

In this tool, we are intending for the exported data to be loaded by a rendering application. Bearing in mind our previous discussion of the non-split BSP tree option, we first specify that we would like to compile a standard tree in which the traditional *split* polygon fragments are stored within its leaves. Next we specify a reasonable split heuristic value of 3.0 and a maximum splitter sample count of 60 to ensure that the BSP compilation is relatively quick and efficient. Finally we enable the removal of any polygon fragments that end up in what should be solid space during the BSP compilation process, by setting the 'm_OptionsBSP.RemoveBackLeaves' variable to true.

We finish initializing the compiler process options variables by finally enabling the T-junction repair process. This module has no further options.

The final part of the constructor is responsible for setting the remaining member variables to their initial states. This includes setting any pointer variables to NULL to ensure that we do not attempt to access an invalid memory location at some future point, as well as setting our compiler's current status variable to that of an idle state.

CCompiler::SetFile

This function is called in order to inform the compiler of the path and filename to the IWF formatted scene file that will be compiled. As we observed in our coverage of the application front end, the 'SetFile' function should be called by the application prior to beginning the actual process of compiling the scene data.

In this function we begin by releasing any string data found to already exist within the 'm_strFileName' variable. This prevents any previously allocated memory from being leaked when we overwrite the pointer value stored in this member in the next step – which is to duplicate the file name string passed as the single parameter to this function. This duplication is achieved by calling the '_tcsdup' function which returns a pointer to a newly allocated copy of the specified string.

```
void CCompiler::SetFile( LPCTSTR FileName )
{
    // Release any old filename
    if ( m_strFileName ) free( m_strFileName );
    m_strFileName = NULL;

    // Duplicate the filename
    m_strFileName = _tcsdup( FileName );

}
```

> **Note:** Remember that when releasing any memory allocated through a call to this string duplication routine, we must use the 'free' function rather than using the more common 'delete' method.

Once we have a copy of this file name string, the compiler will be able to read the contents of the 'm_strFileName' member variable at a later stage in order to determine which file to import. We duplicate this character string, rather than simply storing the specified pointer, because we are unable to guarantee that the string passed to the 'FileName' parameter will not go out of scope and be released before the compiler has had a chance to import the file.

CCompiler::SetOptions / GetOptions

As we know, each compilation processing module in this application has its own unique options structure that can be specified by the application in order to alter the way in which each module performs its task. The two public 'SetOptions' and 'GetOptions' methods are the means by which the application can access and set the options structures for each process module available within the compiler. The first of these two functions – 'SetOptions' – accepts two parameters. The first of these parameters accepts a constant value that specifies the compiler process for which the calling function would like to set the options. The available constant values that can be passed into this parameter are found in the 'Common.h' header file and are shown below:

```
#define PROCESS_HSR         0   // Hidden Surface Removal
#define PROCESS_BSP         2   // Binary Space Partition
#define PROCESS_PRT         3   // Portals
#define PROCESS_PVS         4   // Potential Visibility Set
#define PROCESS_TJR         5   // T-Junction Repair
```

As you can see, there are six constant values defined in this list that match up with each of the processor modules described earlier in this section. Only three of these modules are applicable to this lab project however – 'PROCESS_HSR' for hidden surface removal, 'PROCESS_BSP' for BSP compilation and 'PROCESS_TJR' for the T-junction repair process. The remaining two process modules will be discussed and implemented in the next lesson. This demonstrates however, that each process has a

unique index value that can be used to identify a particular compiler module when interacting with the compiler interface through functions such as those outlined here.

The second parameter declared by this method is a void pointer named ‘Options’, into which the calling function should pass a pointer to the options structure it would like to set. This structure should be of a type applicable to the process specified by the ‘Process’ parameter because this information will be used to cast the void pointer to the correct structure type before assigning that cast structure to the appropriate compiler member variable.

```
void CCompiler::SetOptions( UINT Process, const LPVOID Options )
{
    switch (Process)
    {
        case PROCESS_HSR:
            m_OptionsHSR = *((HSROPTIONS*)Options);
            break;

        case PROCESS_BSP:
            m_OptionsBSP = *((BSPOPTIONS*)Options);
            break;

        case PROCESS_TJR:
            m_OptionsTJR = *((TJROPTIONS*)Options);
            break;

    } // End Switch
}
```

The ‘GetOptions’ method functions in a similar manner to that of the ‘SetOptions’ method. This function accepts the same two parameters. The only difference between these two functions is that the ‘Options’ parameter is now used to pass information back *out* to the calling function, rather than to receive it. Similar restrictions apply to the type of the structure passed to the ‘Options’ parameter. This should match the appropriate options structure type for the specific module passed to the ‘Process’ parameter as before. In the following code block you can see how the void pointer specified by the ‘Options’ parameter is cast to the appropriate type – based on the value contained in the ‘Process’ function parameter – and the data contained within the relevant class member variable is copied into that structure owned by the calling function.

```
void CCompiler::GetOptions( UINT Process, LPVOID Options ) const
{
    switch (Process)
    {
        case PROCESS_HSR:
            *((HSROPTIONS*)Options) = m_OptionsHSR;
            break;

        case PROCESS_BSP:
            *((BSPOPTIONS*)Options) = m_OptionsBSP;
            break;

        case PROCESS_TJR:
            *((TJROPTIONS*)Options) = m_OptionsTJR;
```

```
            break;

    } // End Switch
}
```

CCompiler::PauseCompiler / ResumeCompiler / StopCompiler

One of the features of the compiler core we are developing in lab project 16.2 is its additional support for use in a multi-threaded environment. During the earlier coverage of the 'm_Status' member variable declared by this class, we discussed the fact that the compiler class exposed the ability for the application to alter the state of the compiler object by pausing, resuming or cancelling the current operation. The three methods outlined here are the means by which the application can control this behavior.

The first of these is the 'PauseCompiler' method. In this function we begin by testing the current status of the compiler object. Due to the fact that the act of pausing the process implies that it is already running, we return immediately of the current status variable contains any value other than 'CS_INPROGRESS'. If this test passes, we then update the value of the 'm_Status' variable such that it now represents the paused state. The final task undertaken by this function is to notify the user of the update to the current compilation module's status. This is done by printing a 'Pausing…' message to the user with a call to the logging interface' 'LogWrite' method.

```
void CCompiler::PauseCompiler( )
{
    if ( m_Status != CS_INPROGRESS ) return;
    m_Status = CS_PAUSED;

    // Write Log Info
    if ( m_pLogger )
    {
        // Rewind and write 'pausing' message
        m_pLogger->Rewind( m_CurrentLog );
        m_pLogger->LogWrite( m_CurrentLog, LOGF_WARNING | LOGF_ITALIC, false,
                             _T("Pausing..."));

    } // End if Logger Available
}
```

One thing that should be clear about this function is that it does nothing more than essentially update the compiler's 'm_Status' member, followed by the output of a notification message to the user. At no stage however, does it actually pause any currently executing compilation process. This is due to the fact that each individual module is responsible for testing the current state of the compiler and taking any appropriate action based on that information. The majority of this is achieved through a call to one of the additional compiler functions named 'TestCompilerState' method that we shall be covering shortly.

The next status function available to the application is the 'ResumeCompiler' method. Applicable only if the compiler is currently in a paused state, this function updates the current status of the compiler object only if this is found to be the case. As with the 'PauseCompiler' function previously outlined, this function finally also prints an updated status message to the log channel of the currently executing

process. This function is not responsible for physically resuming the execution of the active compiler process. As before, this is the responsibility of each individual processing module and the 'TestCompilerState' method covered shortly.

```
void CCompiler::ResumeCompiler( )
{
    if ( m_Status != CS_PAUSED ) return;
    m_Status = CS_INPROGRESS;

    // Write Log Info
    if ( m_pLogger )
    {
        // Rewind and write 'resuming' message
        m_pLogger->Rewind( m_CurrentLog );
        m_pLogger->LogWrite( m_CurrentLog, LOGF_WARNING | LOGF_ITALIC, false,
                             _T("Please Wait..."));

    } // End if Logger Available
}
```

The final of the three status update functions is the 'StopCompiler' method. This can be called by the application in order to have the currently executing compiler process cancelled. As with each of the previous two methods, this function is only responsible for updating the 'm_Status' member variable and notifying the user of this updated status. In this case, a value of 'CS_CANCELLED' is used to update the status member only if the compiler process is not idle – e.g. is currently equal to CS_PAUSED or CS_INPROGRESS.

```
void CCompiler::StopCompiler( )
{
    if ( m_Status == CS_IDLE ) return;
    m_Status = CS_CANCELLED;

    // Write Log Info
    if ( m_pLogger )
    {
        // Rewind and write 'pausing' message
        m_pLogger->Rewind( m_CurrentLog );
        m_pLogger->LogWrite( m_CurrentLog, LOGF_WARNING | LOGF_ITALIC, false,
                             _T("Cancelling..."));

    } // End if Logger Available
}
```

With each of these three methods it is important to bear in mind that, while the compiler object's status variable is updated immediately, the currently executing compilation module may not be in a position to respond to this update until a much later point in time. In the interests of efficiency, each module implemented by this lab project will only test the state of the compiler at periodic intervals. This periodic testing can unfortunately lead to a slight delay in which the application must sometimes wait.

CCompiler::TestCompilerState

There are several calls to the 'TestCompilerState' method scattered throughout each of the compiler modules developed in both this lesson and the next. This function is called by the various compiler modules in order to respond to any compiler status change that may have occurred.

When this function is called by the compiler module, the first part of this function deals with the situation in which the compiler has been put into a paused state. As we can see in the following block of code; after informing the user that the compiler has been successfully placed into a paused state this function enters a while loop that will only exit following a call to 'ResumeCompiler' method that will return the 'm_Status' variable to a state of 'CS_INPROGRESS'. Inside this continuing loop, a call is made to the runtime 'sleep' function in order to have the compiler thread wait 500 milliseconds before testing the 'm_Status' member again. By calling the sleep function in this way, we prevent this compiler thread from consuming a great deal of the CPU resources available to our application.

```
bool CCompiler::TestCompilerState( )
{
    if ( m_Status == CS_PAUSED )
    {
        // Write Log Info
        if ( m_pLogger )
        {
            // Rewind and write 'pausing' message
            m_pLogger->Rewind( m_CurrentLog );
            m_pLogger->LogWrite( m_CurrentLog, LOGF_WARNING | LOGF_ITALIC, false,
                                 _T("Paused..."));

        } // End if Logger Available

        // Sleep the thread until compiler is resumed
        while (m_Status == CS_PAUSED) Sleep(500);

    } // End if Paused
```

Assuming the application had not been placed into a paused state, it is possible that the compile process had been cancelled. If this is found to be the case, a value of 'false' is returned to the compiler module making the call to this function in order to inform it of the fact that it should clean up and exit gracefully. In all other cases this function simply returns a value of true to denote that the compile process should continue to execute.

```
    // Should we cancel ?
    if ( m_Status == CS_CANCELLED ) return false;

    // Continue running.
    return true;
}
```

CCompiler::CompileScene

The 'CompileScene' function is the means by which the application can instruct the compiler to execute each of the various compilation modules that may have been enabled. This function should be called by the application only after having selected and informed the compiler of the file that is to be imported and

processed. If any customization of the various module options is required, these should also have been set prior to this function being called. This function accepts no parameters, and returns a single Boolean result that signifies whether the compilation process as a whole was successful or not.

Although not strictly necessary for this method to function correctly, the first task undertaken here is to retrieve the filename portion of the path string stored in the 'm_strFileName' member variable. With the path stripped from the import file string; this will be used only as part of the information output to the compile process log. This is achieved by searching for the *last* forward or backslash contained in the string and – if one exists – retrieving a pointer to the portion of the string which immediately follows that character.

> **Note:** In order to prevent any obfuscation of the logic contained within this and other class methods, much of the logging code has been omitted and replaced with a single line comment outlining the information that would be output to the log at that point. Refer to the source code in the lab project archive provided with this lesson if you wish to view the source files with all log output code included.

```
bool CCompiler::CompileScene( )
{
    CFileIWF IWFFile;

    // Validate Data
    if (!m_strFileName) return false;

    // Retrieve the filename portion only of the string
    LPTSTR FileName = NULL;
    FileName = _tcsrchr( m_strFileName, _T('\\') );
    if (!FileName) FileName = _tcsrchr( m_strFileName, _T('/') );
    if (!FileName) FileName = m_strFileName;
    if (FileName[0] == _T('\\') || FileName[0] == _T('/')) FileName++;
```

The first real job that the 'CompileScene' function must perform is to import the scene information from the IWF file specified by the application. In previous lab projects we have used the 'CFileIWF' class provided by the IWF import library to perform the file handling and data loading steps. Our pre-processing tool is no different in this regard. At the start of this function we have defined an object named 'IWFFile' which is of the type 'CFileIWF'. Although we are using a custom IWF loading class in this application rather than that provided by the libIWF library, the concepts are the same in both cases. As we have done many times in the past, we instruct the IWF loading object to import the data from the IWF file by calling the 'CFileIWF::Load' method, passing in the name and path of the scene data file to be imported as the single parameter input to this function.

```
    try
    {
        // We Are starting the process
        m_Status = CS_INPROGRESS;

        // Load the specified file
        IWFFile.Load( m_strFileName );
```

As has been the case with the 'CFileIWF' class used in each of our previous lab project applications, this class maintains a list of the individual pieces of information loaded from the file within internal STL

vector members. In our rendering applications, it has traditionally been the responsibility of the 'CScene' class to then iterate through these arrays and process each piece of information in an effort to build renderable data where applicable. However, our pre-processing tool is not required to perform any kind of rendering; we simply need to store this information such that we can gain easy access to the file information at any point in the future.

Because of the fact that the STL vector variables within the 'CFileIWF' class are defined with an identical type to those corresponding members contained within our compiler class, making a copy of this data is made extremely simple. As a result of this typed similarity, we can simply take ownership of the information stored in the 'CFileIWF' class instance by assigning the matching member variables in our compiler class to those STL vectors stored in the file object as demonstrated in the following code.

```
        // Obtain ownership of the objects loaded
        m_vpMeshList     = IWFFile.m_vpMeshList;
        m_vpMaterialList = IWFFile.m_vpMaterialList;
        m_vpTextureList  = IWFFile.m_vpTextureList;
        m_vpShaderList   = IWFFile.m_vpShaderList;
        m_vpEntityList   = IWFFile.m_vpEntityList;
```

At this stage we have made a copy of the data stored within the IWF file object into our compiler class's internal member arrays. However, recall that these STL vectors simply store pointers to heap allocated storage classes. This means that by making a simple shallow copy of the array from one class instance to the other, both sets of arrays will now contain elements that reference the same instance of each data object. If the IWFFile object was to go out of scope and be released at some point in the future, this would have the unfortunate side effect of releasing all of the objects now referenced by the compiler class too. To prevent this, we must clear the IWF object's STL vector members to ensure that the compiler object remains the only place in which these data items are referenced.

```
        // Clear out the IWF's object vectors (don't destroy)
        IWFFile.m_vpMeshList.clear();
        IWFFile.m_vpMaterialList.clear();
        IWFFile.m_vpTextureList.clear();
        IWFFile.m_vpShaderList.clear();
        IWFFile.m_vpEntityList.clear();

        // Log - General: 'FileName' was imported succesfully

    } // End try Block
```

Notice in the previous code how the file loading logic was wrapped in a try block that will attempt to catch any exceptions which may have been thrown by the 'CFileIWF::Load' function, or any of the subsequent STL vector operations. The following catch blocks are provided to ensure that all memory allocated to date by both the IWF object during the loading procedure, in addition to any memory allocated by the compiler class are fully released before we return a failure code from this function.

```
    catch (HRESULT & e)
    {
        // Log - General: Import of 'FileName' failed with error 'e'
```

```
        IWFFile.ClearObjects();
        Release();
        return false;

    } // End Catch Block
    catch (...)
    {
        // Log - General: Import of 'FileName' failed

        IWFFile.ClearObjects();
        Release();
        return false;

    } // End Catch Block
```

At this point in the execution of the 'CompileScene' function, we have imported all of the applicable scene data from the source IWF file and have stored that information within the internal STL vector members ready for processing.

We briefly touched upon the fact that the compiler class is responsible for preparing the input data for each process earlier in this section. Rather than have the 'CompileScene' function perform each of these process specific tasks, the code for the actual preparation and execution of each processing module has been placed in a series of functions named 'PerformHSR', 'PerformBSP' and 'PerformTJR'. We will examine these functions in detail shortly, but for now we need only be aware that when any one of these three functions is called, that particular compiler process will be executed from start to finish before returning control back to this 'CompileScene' function.

In the following code we can see that this is the point at which we begin to actually process the scene data, calling each of these three processing functions in order if they have been enabled by the application. In later lessons we will be adding additional compiler modules to the pre-processing tool. For each module that we add, a new separate 'Perform*' method will need to be developed and called from within this function in a similar fashion to those shown below. Once each of the enabled processing functions has been called, we simply exit from this function and return control back to the application having completed the entire compilation process.

```
    // Log - General: Beginning compile run

    // Start compiling by removing all hidden surfaces
    m_CurrentLog = LOG_HSR;
    if ( m_OptionsHSR.Enabled && m_Status != CS_CANCELLED) PerformHSR();

    // Build the BSP Tree if requested
    m_CurrentLog = LOG_BSP;
    if ( m_OptionsBSP.Enabled && m_Status != CS_CANCELLED) PerformBSP();

    // Repair any T-Juncs if requested
    m_CurrentLog = LOG_TJR;
    if ( m_OptionsTJR.Enabled && m_Status != CS_CANCELLED) PerformTJR();

    // Clean up if required
    if ( m_Status == CS_CANCELLED ) Release();
```

```
    // Processing Run Completed
    m_Status = CS_IDLE;

    // Log – General: Compilation Success

    // Success!
    return true;
}
```

As you can see, this relatively simple function is essentially just a wrapper that first loads the scene information, and then simply triggers each of the requested processing modules. Let us therefore begin to take a look at how each of these processing modules is implemented within lab project 16.2.

CCompiler::PerformHSR

The first of the three main module functions implemented within this initial pre-processing application is the 'PerformHSR' function. This function is intended to prepare the data destined to be merged together by the hidden surface removal processor. Recall that the hidden surface removal process accepts a series of meshes that will be combined using the constructive solid geometry *union* operation. This is done in order to remove any surfaces that would introduce illegal geometry into the tree during BSP compilation. If we do not remove these illegal polygon fragments before the tree is compiled, then we end up in a situation in which the solid and empty leaf information is compromised and cannot be resolved. For this reason, the hidden surface removal process must be executed prior to compiling the final scene BSP tree. We will cover the actual hidden surface removal processing module a little later, but first we must understand the type of data that is to be passed into this process and how it is accessed.

In summary, the general process that the 'PerformHSR' function must follow is listed below:

1) Set up the hidden surface removal processor module ready for its execution.

2) Collect a list of all of the meshes in the scene that are not flagged as detail objects and pass each of them into the HSR module class ready for processing.

3) Trigger the HSR module and allow it to run its course.

4) If the HSR process was successful, each of the original non-detail meshes will now have been merged into one single resulting mesh with all hidden surfaces removed. This function must therefore release each of these original meshes from the compiler's scene database in order to ensure that the scene polygon data is not duplicated.

5) Finally, we must retrieve the single resulting mesh from the processing object that constitutes the union of each of those meshes used in the merging operation. This mesh will then be stored in the compiler's internal mesh array ready for compilation by the following BSP compilation process should it have been enabled.

It should be clear from this general outline that this function is not physically responsible for performing any part of the hidden surface removal process. Under the conditions set forth by our modular design,

each processing module is a stand alone class that has its own API. This interface is used by the compiler class in order to initialize, execute and retrieve any data that may have resulted from that module class's execution. While each processing class may differ in the way data is passed in and retrieved, this separation of functionality into individual processing classes should allow us to more easily integrate additional processing tasks into the tool we are building.

We made mention in the previous summary outline that a mesh could be flagged as a 'detail object'. Although we have not been introduced to this type of object in the past you can think of them as being somewhat like those objects that have been linked to the previous spatial tree types using the detail area concept. These are objects that are not compiled directly into the tree itself, but are instead linked at runtime by passing them through the tree and attaching them to any leaves into which they may fall. In this pre-processing tool we treat these objects in a similar fashion. Although the objects will eventually be linked to the leaves only in our rendering application, we must ensure that any scene meshes that have been flagged as detail objects are not altered or compiled into the resulting BSP leaf tree.

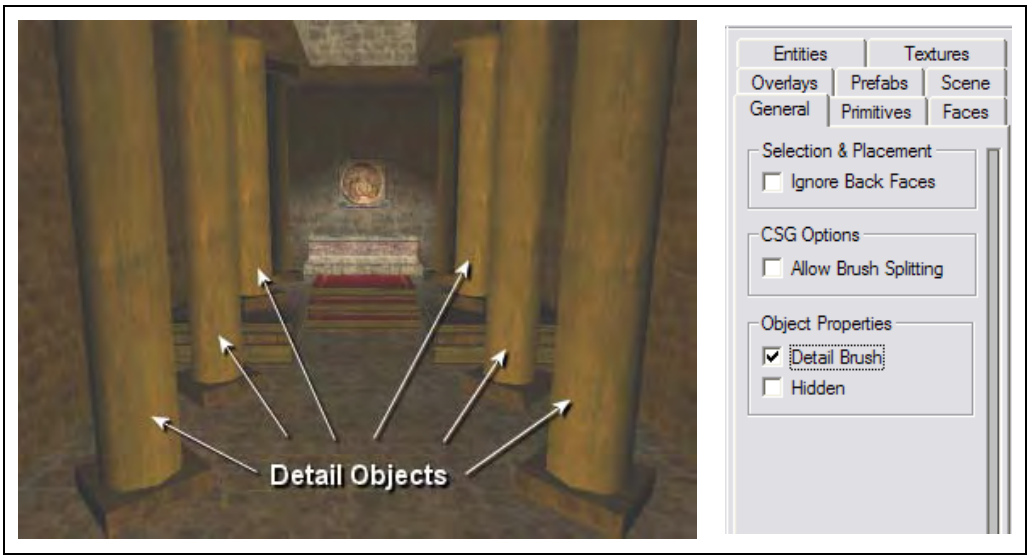


Figure 16.4

Figure 16.4 demonstrates the type of situation in which we might decide to make use of the detail object concept when compiling a level using the polygon aligned BSP leaf tree compiler. This scene contains a series of high resolution cylinders used to depict several roof support columns. Objects – such as these cylinder meshes – which contain a high level of detail, or a large amount of variation in the orientation of their polygons, can be ideal candidates for selection as detail objects. This is due to the fact that objects of this type can potentially introduce many hundreds of nodes and leaves when included in the BSP tree compilation process, without providing any significant benefit to the tree structure. In this figure we can also clearly see that these column meshes would not provide a significant amount of occlusion information for the visibility module (developed in the next lesson) that might have made their inclusion in the tree worthwhile.

On the right hand side of this figure we have included an image of the general properties tab from the GILES™ level editing tool. We can use this editor to select any mesh in our IWF scene, and flag it to become a detail object by checking the 'Detail Brush' box highlighted. After saving scene back out to

file, when we subsequently load it back into our rendering application or compiler tool we can check for the existence of this flag using the following bit test:

```
if ( pMesh->Flags & MESH_DETAIL ) { // This is a detail object }
```

Another good example of common types of detail objects are those meshes that are used as set decoration within the scene. This might include items such as tables, chairs, beds or basically anything that is not part of the scene architecture.

So, armed with this additional knowledge of what types of meshes should and should not be included in the hidden surface removal and BSP compilation processes, let us now take a look at how the 'ProcessHSR' function has been implemented in this lab project.

```
bool CCompiler::PerformHSR()
{
    // One time compile process
    CProcessHSR ProcessHSR;
    ULONG       i, MeshCount = 0;

    // Set our processor options
    ProcessHSR.SetOptions( m_OptionsHSR );
    ProcessHSR.SetLogger( m_pLogger );
    ProcessHSR.SetParent( this );

    // Log – HSR: Beginning Process
```

As we should have gathered from our examination of 'CompileScene', this function accepts no parameters. All of the data required by this function should have been imported into the compiler class' internal member arrays at this point, and can be access directly within this function. At the top of this function we can observe that an instance of our first processing class 'CProcessHSR' is created on the stack, as a local variable named 'ProcessHSR'. This object exposes all of the functionality required for initializing, executing and retrieving the mesh data resulting from the hidden surface removal process.

Once this processing object has been instantiated, we can then begin to initialize the various settings and values required by this module. The first of these is the 'HSROPTIONS' structure that we encountered earlier. Like the compiler class itself, each of the processing module classes – such as 'CProcessHSR' – implements a 'SetOptions' function used to set that object's option values. Recall that the 'HSROPTIONS' structure contains only a single variable that denotes whether or not this process is enabled. Regardless of this fact, we pass the options structure in at this point in order to ensure that the hidden surface removal class will have access to any additional process settings that may be added to the options structure in the future.

During each of the processing tasks, specific progress and status information must be reported to the user as we know. The next item that we pass to the HSR processing object therefore, is a reference to the logging class specified by the application. This gives the module a chance to report any process specific failures that may have occurred, in addition to providing any general information such as an operation progress percentage.

The final item that we pass into the 'ProcessHSR' object in the previous block of code is a reference to the compiler object itself. Having access to the compiler object will allow the process module to call the aforementioned 'TestCompilerState' method of the compiler as required.

With the processing module having been initialized with all of the various options and house-keeping items we can now begin to pass in the mesh data that will be used. Before we do this however, we must perform a final validation step. Of course, in order to perform a union CSG operation such as that required by the hidden surface removal process, we need at least *two* meshes in order for this operation to function correctly. If there is only one mesh in the scene to begin with, then there would be nothing else available for us to merge that mesh with. With this in mind, we first total the number of non detail meshes that have been loaded. If we find ourselves in a situation whereby there is only one available scene mesh to be processed and compiled, we simply output this information to the user and take no additional action.

```
    // Count the number of scene meshes
    for ( i = 0; i < m_vpMeshList.size(); i++ )
    {
        CMesh * pMesh = m_vpMeshList[i];
        if ( !pMesh ) continue;

        // Tally Mesh ?
        if ( !(pMesh->Flags & MESH_DETAIL) ) MeshCount++;

    } // Next Mesh

    // How many meshes ?
    if ( MeshCount <= 1 )
    {
        // Log – HSR: Nothing to do.

    } // End if single mesh
    else
    {
```

At this stage we know that we have at least two non detail meshes that must be unioned in preparation for the BSP compilation step. In the next section of code from this function we then begin to loop through each of the meshes stored in the compilers internal mesh array. If there is no mesh stored in any particular element (e.g. the mesh pointer stored there is equal to NULL) then we automatically continue to the next element in the array. It is imperative that we perform this simple test due to the fact that whenever a mesh is to be removed from consideration it is simply released before having its associated element in the mesh array set to NULL. This is done in the interests of efficiency and prevents us from having to reshuffle the mesh array at each step. We will see this behaviour shortly. In addition to testing the mesh pointer, the content of the mesh's 'Flag' member must also be tested within this loop. If this mesh is considered to be a detail object, then it should not be merged by the HSR process and so we simply move on to the next element in the array without action. With both of these tests completed, we reach a point in which we have a valid pointer to a non-detail mesh. This mesh is then passed into the HSR processing object with a call to the 'AddMesh' method of that class.

Once this loop has completed, the HSR processing module will now have a list of every mesh that we would like to merge together into one single resulting mesh, with all illegal geometry having been

removed. This is all the information this processing module needs to be aware of and so we call the object's 'Process' method to instruct the module to perform the actual hidden surface removal step on those meshes specified.

```
        // Add all of our meshes to the HSR Processor
        for ( i = 0; i < m_vpMeshList.size(); i++ )
        {
            CMesh *pMesh = m_vpMeshList[i];
            if (!pMesh) continue;
            if ( pMesh->Flags & MESH_DETAIL ) continue;
            ProcessHSR.AddMesh( pMesh );

        } // Next Mesh

        // Begin the HSR Process
        ProcessHSR.Process();
```

When the module's 'Process' function returns, it will have constructed a new mesh object that contains each of the polygons having been merged during the hidden surface removal process. This is of course unless an error had occurred. While the HSR processing class is responsible for reporting the specifics of any errors to the user via the logging class specified, we must still ensure that we take the appropriate action. To determine whether or not an error occurred during the hidden surface removal process, we test the value returned by the modules 'GetResultMesh' function. If this returns a value not equal to NULL then we know that no error occurred and we can continue to process the resulting data.

Before we actually do anything with the mesh generated by the 'Process' method, we must first remove each of the original non-detail objects that were unioned together in this step. As mentioned earlier, we do this in order to ensure that the final scene data set does not contain duplicate information. This is achieved using a method similar to that of the previous loop in which we added meshes to the processor prior to its execution. In this case however, whenever we find a valid pointer to a non detail mesh in the compiler's mesh array, we release that mesh object. Recall that we validated each element in the mesh array earlier to ensure that it did not contain a NULL pointer. The following loop logic demonstrates the reason why this must be done – once the original mesh object has been released, we subsequently set the corresponding array element to NULL. These empty elements will be removed at a later point in the application but at this stage this is done simply to prevent the mesh array from being reshuffled and resized at each step.

```
        // Make way for our result if there is one
        if ( ProcessHSR.GetResultMesh() != NULL )
        {
            // Destroy any loaded meshes
            for ( i = 0; i < m_vpMeshList.size(); i++ )
            {
                CMesh * pMesh = m_vpMeshList[i];
                if ( !pMesh ) continue;
                if ( pMesh->Flags & MESH_DETAIL ) continue;
                delete pMesh;
                m_vpMeshList[i] = NULL;

            } // Next Mesh
```

With each of the original source meshes removed, we can now retrieve the resulting mesh from the 'ProcessHSR' object and store it in the compiler's mesh array ready for the next compilation process. Rather than simply add this mesh to the end of the mesh array however, we have the ability to reuse one of the array elements that we had set to NULL in the previous loop. The following code searches for an array element containing a NULL pointer. If one is found, the index to that element is stored in the local 'MeshIndex' variable before breaking out of the loop.

```
            // Store our single result mesh in the first empty slot
            long MeshIndex = -1;
            for ( i = 0; i < m_vpMeshList.size(); i++ )
            {
                if ( !m_vpMeshList[i] ) { MeshIndex = i; break; }

            } // Next Mesh Slot
```

Notice that the local variable 'MeshIndex' was initialized using a default value of -1 in the previous code. If there were no elements found to contain a NULL pointer, this variable will not have been overwritten with a new index value and should therefore still be in its default state of -1. If an empty slot in the mesh array *was* found, this variable would now contain an index in the range of 0 and above. The following code examines the contents of this variable to determine whether it can reuse one of the available empty array elements, or if it must add the new resulting mesh to the end of the compiler's internal mesh array. If the 'MeshIndex' array contains a value of -1 then no slot was found and a pointer to the new mesh – accessed with a call to the module's 'GetResultMesh' method as before – is added to the end of the mesh array with a call to the STL vector 'push_back' function. If on the other hand the value of 'MeshIndex' is anything other than -1, then we simply overwrite that element in the array with the new mesh pointer.

```
            if ( MeshIndex == -1 )
            {
                m_vpMeshList.push_back( ProcessHSR.GetResultMesh() );

            } // End if No Slot
            else
            {
                m_vpMeshList[ MeshIndex ] = ProcessHSR.GetResultMesh();

            } // End if Free Slot
```

At this stage we have now removed every non-detail mesh originally contained in the compiler's mesh array, merged them together, and finally replaced them with a single new mesh result. With all of the illegal geometry hopefully now removed from the scene data set, we are now ready to have this new mesh processed by the next module in the list – the BSP compiler. As a result, the last part of this function simply informs the user that the HSR process has completed, and then returns to the calling 'CompileScene' function.

```
        } // End if

    } // End if multiple meshes
```

```
    // Log – HSR: Write Success Information

    // Success!!
    return true;
}
```

Although we have not yet covered the code to the hidden surface removal processor at this point, the discussion of this function will serve us well in understanding how this module fits into the larger compilation scheme. In addition to this we should now have a clearer picture of exactly what data is passed to the module class and what pieces of information the compiler is expecting to receive.

CCompiler::PerformBSP

Referring back to the code in the 'CompileScene' function, we know that this 'PerformBSP' function is called directly following the hidden surface removal step. With the HSR process completed, we are now able to compile a solid BSP leaf tree without worrying about the impact that any illegal geometry might have on the validity of the tree structure generated.

This is the stage in which we begin constructing the data that will serve as the foundation for much of the additional information generated by the future modules we will be implementing in both this lesson and the next. As we know, the 'CCompiler' class contains a member variable named 'm_pBSPTree'. This variable will store the pointer to an instance of the 'CBSPTree' class we will be discussing shortly. In the next lesson we will be using much of the information constructed in this step to generate the portal data that describes how the leaves in the tree are connected together. These portal polygons will then allow us to build the inter-leaf visibility information (PVS) which will provide us with scene occlusion culling. All of this information will eventually be stored within the BSP tree we construct in this step which will also be used as the source from which the final scene file is built.

From a high level standpoint, the 'PerformBSP' function undertakes the following tasks:

1) Set up the BSP compiler module ready for its execution.

2) Collect a combined list of all of the polygons from every non detail mesh in the scene and pass that list to the BSP tree compiler. These polygons will be used as the basis for the compilation of the polygon-aligned BSP leaf tree itself.

3) Trigger the BSP compilation process and allow it to run its course.

4) If the BSP compile process was successful, any original non-detail meshes that were used in the construction of the BSP tree should be removed. The polygon data is now owned by the centralized scene BSP tree stored within the compiler object. When exporting the scene at the end of the compilation process, the polygon data will be built using that information contained within the BSP tree.

As we can see from the above overview, the responsibilities of this method are fewer in number than those of the 'PerformHSR' function. We should already be familiar with the types of data required by

the BSP compiler at this point, so we will get straight down to examining the implementation of the 'PerformBSP' function.

```
bool CCompiler::PerformBSP()
{
    ULONG       i;

    // Destroy any old BSP Tree
    if ( m_pBSPTree ) delete m_pBSPTree;

    // Allocate a new BSP Tree
    m_pBSPTree = new CBSPTree;

    // Set our compiler options
    m_pBSPTree->SetOptions( m_OptionsBSP );
    m_pBSPTree->SetLogger( m_pLogger );
    m_pBSPTree->SetParent( this );

    // Log – BSP: Beginning Compilation Process
```

In a similar fashion to the 'PerformHSR' method, there are no parameters that need to be specified when calling this function. As before, everything we need is already contained within the member variables of the compiler class itself. Unlike this previous method however, there is no local variable declared here which serves as the processing module. In the case of hidden surface removal, the processing module class we used was instantiated as a temporary object which simply processed and built data subsequently used as a replacement in one single step. With the BSP tree class, we are building a tree structure that must be retained both for future processing modules, in addition to the requirement that this structure to be exported to file at a later point in the application. As a result, we create an instance of the 'CBSPTree' class and store it within the 'm_pBSPTree' member variable of the compiler class.

After creating the BSP class instance, we specify the compilation options that it must use with a call to the 'CBSPTree::SetOptions' method, passing in the 'BSPOPTIONS' structure stored in the compiler's 'm_OptionsBSP' member variable. We also inform the compiler module of the applications logging class object, in addition to passing the reference to this compiler object with calls to the 'SetLogger' and 'SetParent' calls respectively.

With the main BSP tree member object instantiated and initialized, we are now ready to begin processing the loaded scene information in order to compile the tree structure. In the following code we begin by looping through each of the meshes currently stored within the compiler's mesh array, adding each mesh's polygon data to the tree ready for compilation. If the hidden surface removal process was enabled and executed in the previous step, it is possible that this array will only contain a single valid mesh pointer. With that said, it is not a requirement that the HSR process be enabled in cases where the scene contains no illegal geometry to begin with, or those in which valid solid / empty information is not required. For this reason we must still loop through each mesh rather than simply searching for the first valid mesh pointer. Also recall that we do not want to include those meshes flagged as being detail objects in the compilation of the BSP tree. To this end we skip over any meshes whose 'Flag' variable contains the 'MESH_DETAIL' bit as before. Each time we find a mesh in the array that should be included in the BSP compilation process, we add its polygon data to the BSP compiler object with a call to the 'AddFaces' method. This function accepts two parameters. The first parameter should be passed

an array of ‘CFace’ object pointers – such as those managed by each mesh – each of which will be appended to the internal list of polygons to be used. The second of these two parameters should be passed a simple integer value containing the length of the aforementioned array.

With each mesh’s polygon data passed into the BSP tree object, the process if compiling the tree can now begin. In the case of the ‘CBSPTree’ class, this is done with a call to its ‘CompileTree’ method.

```
    // Add all remaining meshes polys (automatically backup with NSR)
    for ( i = 0; i < m_vpMeshList.size(); i++ )
    {
        CMesh * pMesh = m_vpMeshList[i];
        if ( !pMesh ) continue;
        if ( pMesh->Flags & MESH_DETAIL ) continue;
        m_pBSPTree->AddFaces( pMesh->Faces, pMesh->FaceCount );

    } // Next Mesh

    // Compile the BSP Tree
    m_pBSPTree->CompileTree();
```

Once the BSP tree has been successfully compiled, each of the non-detail meshes used to construct that tree should now be removed from the compiler’s scene database. If we did not perform this step then our export file would contain at least one duplicate of every polygon in the tree – one stored in the source mesh and another in the BSP tree itself. This loop simply searches for any valid non-detail mesh, releases it and finally overwrites the corresponding element in the compiler’s mesh array with a NULL pointer.

```
    // Destroy any loaded scene meshes
    for ( i = 0; i < m_vpMeshList.size(); i++ )
    {
        CMesh * pMesh = m_vpMeshList[i];
        if ( !pMesh ) continue;
        if ( pMesh->Flags & MESH_DETAIL ) continue;
        delete pMesh;
        m_vpMeshList[i] = NULL;

    } // Next Mesh
```

In the final part of this function we unset the BSP tree object’s logging class reference by passing NULL to the ‘SetLogger’ function. This is not strictly necessary within our lab project application. However, in doing so, we ensure that any BSP tree class function that outputs progress information during the main tree compile step does not output this information again in the event that any future module makes a call to it.

```
    // Prevent future logging
    m_pBSPTree->SetLogger( NULL );

    // Log – BSP: BSP Process Completed

    // Success!!
    return true;
```

```
}
```

With the BSP compilation step completed we once again return control back to the 'CompileScene' function that made the original call to this method. The next processing task undertaken by the compiler is the T-junction repair step.

CCompiler::PerformTJR

The 'PerformTJR' function is the primary wrapper around the preparation and execution of the T-junction repair process. This process constitutes the final pre-processing task that we will develop in this lesson.

The process of T-junction repair has been covered in previous lessons, so we should already be familiar with the general concepts involved. We know from experience that – at a high level – this process compares the vertices of every polygon in the scene against the edges of every other polygon that surrounds it. If no vertex is found to have a matching position within the edge being tested then a copy of that vertex must be inserted in the neighboring edge in order to resolve this T-junction situation. With this in mind, it should be clear that we must somehow pass the scene polygon information into the T-Junction repair module in order for it to have access to the level geometry. At this point in the process however, there are two locations in which the scene polygon information may exist. The first is from the mesh data contained within the compiler's internal mesh array. However, this array will only contain applicable mesh data if the BSP compilation step was bypassed. Recall that during the BSP compile step, the 'PerformBSP' function may remove those non-detail meshes that were used to construct the tree if the BSP process has been enabled by the application. As a result, the second case that we must consider is that in which the BSP tree has already been compiled and we must source the polygon data from the compiler's BSP tree object as a result.

As we know, the T-Junction repair process is not concerned with any of the renderable polygon information such as textures or materials, or even the surface normal. All this process needs to be able to function is a list of the vertices that make up each of the polygons in the scene. If we refer back to the discussion of the base 'CPolygon' class – from which each of the extended polygon data classes are derived – it should be clear that this process need only access the information exposed by that base class.

With this situation in mind, the T-Junction repair processing module has been designed to accept a list of *pointers* to one or more 'CPolygon' objects. In doing so, we allow the T-Junction repair process to gain access to the polygons from a wide variety of sources even if they are of different physical types. As long as each of the source polygon objects to be repaired are *derived* from the base 'CPolygon' class, then we can pass this data into the repair module and have the process function regardless of their type.

Before we move onto examining the internal workings of this method, let us quickly summarize the procedure as it will be implemented.

1) Set up the T-Junction repair module ready for its execution.

2) Collect a combined list of all of the polygons from every non detail mesh in the scene, or alternatively from any compiled BSP tree object, and pass that list to the T-Junction repair module ready for processing.

3) Trigger the repair process and allow it to run its course.

As you can see, this function is a relatively simple one. Let us therefore move on to briefly discuss how it has been implemented in this lab project.

```
bool CCompiler::PerformTJR()
{
    ULONG i, k;

    // One time compile process
    CProcessTJR ProcessTJR;

    // Set our process options
    ProcessTJR.SetOptions( m_OptionsTJR );
    ProcessTJR.SetLogger( m_pLogger );
    ProcessTJR.SetParent( this );

    ULONG       PolyCount = 0, Counter = 0;
    CPolygon ** ppPolys   = NULL;

    // Log – TJR: Beginning Process
```

As with each of the tasks before it, the first thing we must do in this function is to instantiate the T-junction repair module object and initialize the various options, logging and parent properties. Similar to the hidden surface removal process, T-junction repair is a one time procedure that simply manipulates any data that already exists in the scene. Once the process has completed, this processing module object is no longer required. As a result, the 'CProcessTJR' class is instantiated as a local variable named 'ProcessTJR'. Once this object has been initialized we can begin to prepare any applicable scene information ready for the T-junction repair module to perform its task.

The first case we deal with in this function is that in which no BSP tree has yet been compiled. In this case there may be one or more non-detail meshes contained within the compiler's internal scene mesh array. Because we cannot know in advance how many polygons are stored within these mesh objects in total, we must first iterate through each non-detail mesh and count up this total. If it turns out that there are no available polygons in the list, this function can simply return without error.

```
    if ( !m_pBSPTree )
    {
        // Count all polys in all active meshes
        for ( i = 0; i < m_vpMeshList.size(); i++ )
        {
            CMesh * pMesh = m_vpMeshList[i];
            if ( !pMesh ) continue;
            if ( pMesh->Flags & MESH_DETAIL ) continue;
            PolyCount += pMesh->FaceCount;

        } // Next Mesh
```

```
        // If no polys exist, bail (no error)
        if (!PolyCount) return true;
```

Now that we know the total number of polygons contained within each of the meshes in the compiler' scene database, we can begin to allocate and build the polygon pointer array discussed earlier. Our first job therefore is to allocate an array of 'CPolygon' base class *pointers* that will store any references to each polygon needed. This pointer returned by this allocation is stored in the local variable 'ppPolys' declared at the top of the 'PerformTJR' function. If this array was allocated successfully, this function subsequently collects a pointer to each polygon, from every valid non-detail mesh, and stores them in this newly allocated array.

```
        // Allocate our polygon pointer array
        if (!(ppPolys = new CPolygon*[PolyCount])) return false;

        // Add all mesh face pointers here
        for ( i = 0, Counter = 0; i < m_vpMeshList.size(); i++ )
        {
            CMesh * pMesh = m_vpMeshList[i];
            if ( !pMesh ) continue;
            if ( pMesh->Flags & MESH_DETAIL ) continue;

            // Loop through faces
            for ( k = 0; k < pMesh->FaceCount; k++ )
                ppPolys[ Counter++ ] = pMesh->Faces[ k ];

        } // Next Mesh
    } // End if Meshes
```

At this stage, assuming no BSP tree had been compiled prior to this function being called, the local 'ppPolys' variable will reference an array that contains a pointer to every polygon, taken from every non-detail mesh in the scene. If a BSP tree had been compiled however then, as we know, there will be applicable meshes remaining in the compiler's mesh array. As a result, we must also cater for the situation in which a BSP tree has been compiled.

In this case, we already know in advance how many polygons are stored in the already compiled BSP tree object. We can gain access to this information by calling the 'CBSPTree::GetFaceCount' method. Again, if the number of polygons is found to be 0 at this stage, then we return from the function without error. With the total number of polygons to hand, we allocate the 'CPolygon' pointer array on the heap and store its pointer in the local 'ppPolys' array in the same fashion as the previous case. If the allocation of this array was successful, we finish up this conditional block by filling the same polygon pointer array with references to each polygon contained in the BSP tree. These polygons are retrieved by calling the 'CBSPTree::GetFace' method.

```
    else
    {
        // Retrieve poly count
        PolyCount = m_pBSPTree->GetFaceCount();

        // If no polys exist, bail (no error)
```

```
        if (!PolyCount) return true;

        // Allocate our polygon pointer array
        if (!(ppPolys = new CPolygon*[PolyCount])) return false;

        // Loop through faces and add
        for ( k = 0; k < PolyCount; k++ )
            ppPolys[ k ] = m_pBSPTree->GetFace( k );

    } // End if BSP Tree
```

So, in either of the cases outlined above we now have an array that contains a list of polygons, referenced by pointers to their base 'CPolygon' class. This will of course be the case regardless of the actual type of the polygon objects being referenced. Armed with all the information we need, we then trigger the T-Junction repair module with a call to the 'ProcessTJR' object's 'Process' function, passing in both the array we have just constructed, along with the number of polygons stored therein. Due to the fact that we have passed in only *references* to the polygon data contained in either the mesh array, or the BSP tree, the T-Junction repair process will manipulate the polygon data without the need to be aware of the location from which they were retrieved. In addition, because the process processes and inserts vertices directly into the source polygon data, there is no need for us to retrieve and interpret any kind of resulting data. Once the repair process has completed, there is nothing further required of this function other than to release the temporary polygon array that we had previously allocated. With this allocated resource released, we finally report the process success to the user and return control to the calling function.

```
    // Repair any T-Junctions
    ProcessTJR.Process( ppPolys, PolyCount );

    // Release the poly pointer array
    if (ppPolys) delete []ppPolys;

    // Log – TJR: Process Completed

    // Success!!
    return true;
}
```

Once this function has returned, we will have come to the end of the entire compilation process for this lab project. In future lessons we will of course add additional processing modules to the compiler core that will add additional steps. In this version of our pre-processing tool however, the front end application would instruct the compiler to save the resulting information at this point. Therefore, let us now examine the compiler method used to save our newly compiled scene information to file.

CCompiler::SaveScene

The final task that our compiler must undertake once the scene data has been successfully processed is to save the compiled information back out to file.

There are several pieces of information that we have imported, compiled or processed that need to be written back out to the final IWF file. These include the texturing, material, shader and entity information we imported earlier in addition to any surviving mesh data that still exists within the compiler's mesh array. Of course, we must also export the BSP tree data to file too should that compiler process have been enabled. Let us briefly refer back to a statement that was made earlier in this chapter regarding the 'BuildFromBSPTree' function exposed by the new 'CMesh' class.

"Although we have not yet discussed our new BSP leaf tree class, we should be familiar enough with the general concepts involved in the construction of leaf trees to know that, once the compilation has completed, the tree will store a list of each of the polygons contained within its leaves. Once the BSP tree has been built, we need to write the polygon information to file in a format that is easy for the application to retrieve. Due to the fact that our application is already capable of loading mesh and polygon data from the IWF file, it makes sense that we export the resulting BSP tree polygon data out to file as a standard mesh. This will prevent us from having to implement another series of polygon import functionality and will allow the BSP tree's polygon data to be loaded seamlessly by any IWF import procedure."

This behavior can be observed in the function code outlined here. Should the BSP tree have been compiled, we take the polygon data contained within the compiled tree object and convert it into a 'CMesh' object such that it is compatible with both the standard mesh saving and loading procedures. This mesh is then added to the IWF file object in a manner that will cause this mesh to **always** be written to the file as the first mesh encountered during any subsequent import operation. Each mesh that follows this first mesh will be the detail objects that have not been processed or compiled into the BSP tree by this application. Moving forward it will be important to bear this in mind as we begin to develop the rendering application that will rely on this exported IWF file.

With all the various pieces of information stored within the compiler, it may seem as if the process involved in exporting the resulting scene data could be a little daunting. In fact, this process is relatively simple as we will discover when we begin to examine the implementation of the 'SaveScene' function below.

```
bool CCompiler::SaveScene( LPCTSTR FileName )
{
    ULONG       i;
    CFileIWF    IWFFile;
    CMesh      *pTreeMesh = NULL;

    try
    {
        // Fill up the IWF's internal structures
        IWFFile.m_vpTextureList  = m_vpTextureList;
        IWFFile.m_vpMaterialList = m_vpMaterialList;
        IWFFile.m_vpShaderList   = m_vpShaderList;
        IWFFile.m_vpEntityList   = m_vpEntityList;
```

This function accepts only a single parameter to which the application should pass the path and filename to the file in which it would like the current scene data to be saved. As with the loading operation undertaken in 'CompileScene', this function makes use of the 'CFileIWF' class to provide much of the

core file handling and export functionality. All we really need to do at this point is to populate the data structures within this class that it will then proceed to interpret and export to the final IWF file. To this end we begin by creating a local instance of this class in the form of the 'IWFFile' object shown at the top of this function.

Recall in our previous discussion of the 'CompileScene' function, we took ownership of several pieces of the imported data simply by performing a shallow copy on the STL vectors contained within the IWF loading class. During this save operation we can do exactly the same thing in reverse. By assigning the 'IWFFile' object's STL vector members to those matching vectors stored within the compiler object, we populate those data structures automatically. However, due to the fact that we must perform additional processing on the mesh information that is to be exported, we can only do this with the static information that was not modified during the compilation process. These include the texture, material, shader and entity vector members demonstrated in the previous code snippet.

With these other four STL vectors now populated, this function then proceeds to construct the all important mesh and polygon data that must also be exported to the IWF file. Depending on whether or not the application chose to enable the BSP tree compilation step, there are clearly two different situations that can arise. Either a BSP tree object has been created, from which we must populate the export file, or no tree was compiled and we simply need consider those meshes contained within the compiler's mesh array. The first of these two cases is shown below. If a BSP tree has been constructed at this point, the function will proceed to generate a compatible 'CMesh' object from the polygon data contained in the BSP tree. We achieve this by calling the 'BuildFromBSPTree' method of the 'CMesh' class covered earlier. To this function we pass the primary instance of our BSP tree class stored within the 'm_pBSPTree' member variable. Once completed, this newly generated mesh is then added to the IWF object's mesh list in exactly the same fashion as if this had been a standard mesh to begin with. Of course, the additional tree information such as the nodes and leaves will not be transferred into this mesh object. This additional information will be handled separately, as we shall see later on in this function.

```
        // If there is a tree, build a mesh from it
        if ( m_pBSPTree )
        {
            // Allocate a new tree mesh
            pTreeMesh = new CMesh;
            if (!pTreeMesh) throw std::bad_alloc();

            // Build the mesh from the bsp tree
            if (!pTreeMesh->BuildFromBSPTree( m_pBSPTree ))
                throw BCERR_OUTOFMEMORY;

            // Add the mesh to the file export list
            IWFFile.m_vpMeshList.push_back( pTreeMesh );

        }
```

The alternate case is executed when the application chose not to construct a BSP tree from the imported mesh data. In this case we simply add each of the **non-detail** meshes directly to the end of the 'IWFFile' object's mesh list ready for export.

```
        else
        {
            // Add any conventional meshes
            for ( i = 0; i < m_vpMeshList.size(); i++ )
            {
                CMesh * pMesh = m_vpMeshList[ i ];
                if ( !pMesh ) continue;

                // Add this if it's not a detail mesh
                if ( !(pMesh->Flags & MESH_DETAIL) )
                    IWFFile.m_vpMeshList.push_back( pMesh );

            } // Next Mesh

        } // End if No Tree
```

Although it may seem strange that we have chosen to add only those non-detail mesh objects to the export list in the previous case, remember that we must export any detail objects back out to file regardless of whether or not we have compiled a BSP tree. As a result, the detail objects are added to the list outside of the conditional if/else statement to ensure that these detail mesh objects are exported in both cases.

```
        // Loop through and add any detail meshes
        for ( i = 0; i < m_vpMeshList.size(); i++ )
        {
            CMesh * pMesh = m_vpMeshList[ i ];
            if ( !pMesh ) continue;

            // Add this if it's a detail mesh
            if ( pMesh->Flags & MESH_DETAIL )
                IWFFile.m_vpMeshList.push_back( pMesh );

        } // Next Mesh
```

Now that each of the 'IWFFile' object's four internal STL vectors have been populated with the appropriate data items, we can now call the 'Save' function to have that data written to file. To this function we pass both the name of the file to which the scene is to be exported, in addition to a pointer to the BSP tree object stored in the compiler's 'm_pBSPTree' member variable. While the purpose of the first parameter should be obvious, we pass the BSP tree object to this function as the second parameter in order for the IWF class to access and export the additional information stored within the tree. These include the nodes, leaves and plane information that may have been generated during the compilation process. In the next lesson, the tree object will also store portal and visibility information that the 'CFileIWF' class will also be responsible for retrieving and exporting.

Assuming the save operation was a success; we must now clear up any items that may have been allocated during this function's execution. The first of these is the mesh pointed to by the local 'pTreeMesh' variable. Recall that if a BSP tree object was constructed during the compilation process, this function was required to generate a new mesh object that could be added to the standard mesh list within the 'IWFFile' object. If this case has occurred, then we must release this temporary mesh object before exiting the function in order to prevent a memory leak. Finally we clear each of the 'IWFFile'

object's internal STL vector items. We must do this to prevent the objects stored within this array from being released prematurely when the 'IWFFile' object goes out of scope as this function returns.

```
        // Export file
        IWFFile.Save( FileName, m_pBSPTree );

        // Clean up
        if ( pTreeMesh ) delete pTreeMesh;
        IWFFile.m_vpMeshList.clear();
        IWFFile.m_vpTextureList.clear();
        IWFFile.m_vpShaderList.clear();
        IWFFile.m_vpEntityList.clear();
        IWFFile.m_vpMaterialList.clear();

    } // End Try Block
```

This function traps any errors encountered using a try/catch block mechanism. Because an exception may be thrown during either the IWF export procedure, or during any of the STL operations we perform in this function, the catch block below attempts to safely release any allocated memory in an identical manner to that outlined in the earlier code. If an exception was thrown within this function then the code in this catch block will be executed before returning a value of 'false' to the calling function. If no exception is thrown then the catch block is skipped and the function will return a value of 'true' denoting a successful export.

```
    catch (...)
    {
        if ( pTreeMesh ) delete pTreeMesh;
        IWFFile.m_vpMeshList.clear();
        IWFFile.m_vpTextureList.clear();
        IWFFile.m_vpShaderList.clear();
        IWFFile.m_vpEntityList.clear();
        IWFFile.m_vpMaterialList.clear();
        return false;

    } // End Catch Block

    // Success!!
    return true;
}
```

CCompiler::Release / ~CCompiler()

With the scene now fully processed, the only two functions that remain for us to examine are those responsible for cleaning up any objects or resources allocated during the import and compilation procedures. The first of these two functions is the 'Release' method. Similar to the functions of this type that we have implemented in the past, this can be called by the application should it wish to reuse any particular instance of the compiler class in a subsequent compilation task.

There is nothing really deep or complex about this function, the majority of which is dedicated to simply looping through each of the STL vectors maintained by the compiler class, and releasing any data storage objects stored in each element. These include meshes, textures, materials, shaders and entities.

Once each of these objects has been correctly released, the STL vectors in which they were contained are cleared in order to ensure that these arrays are cleared of the object pointers that are now invalid.

```
void CCompiler::Release()
{
    unsigned long i;

    // Destroy any loaded meshes
    for ( i = 0; i < m_vpMeshList.size(); i++ )
    {
        if ( m_vpMeshList[i] ) delete m_vpMeshList[i];

    } // Next Mesh

    // Destroy any loaded textures
    for ( i = 0; i < m_vpTextureList.size(); i++ )
    {
        if ( m_vpTextureList[i] ) delete m_vpTextureList[i];

    } // Next Texture

    // Destroy any loaded maerials
    for ( i = 0; i < m_vpMaterialList.size(); i++ )
    {
        if ( m_vpMaterialList[i] ) delete m_vpMaterialList[i];

    } // End If

    // Destroy any loaded shaders
    for ( i = 0; i < m_vpShaderList.size(); i++ )
    {
        if ( m_vpShaderList[i] ) delete m_vpShaderList[i];

    } // End If

    // Destroy any loaded entities
    for ( i = 0; i < m_vpEntityList.size(); i++ )
    {
        if ( m_vpEntityList[i] ) delete m_vpEntityList[i];

    } // End If

    // Clear the vectors
    m_vpMeshList.clear();
    m_vpTextureList.clear();
    m_vpMaterialList.clear();
    m_vpShaderList.clear();
    m_vpEntityList.clear();
```

With each of the scene data items released and cleared, the final task is to release the duplicated file name string contained in the ‘m_strFileName’ variable and to release the compiled BSP tree object should it exist.

```
    // Release strings
```

```
    if ( m_strFileName ) free( m_strFileName );
    m_strFileName = NULL;

    // Destroy any compiled BSP Tree
    if (m_pBSPTree) delete m_pBSPTree;
    m_pBSPTree = NULL;
}
```

The second of these two functions is the standard compiler class destructor. As has been the case in many of the classes we have encountered, this destructor simply calls the 'Release' function to ensure that no resources are leaked when the compiler class is destroyed or goes out of scope.

```
CCompiler::~CCompiler()
{
    // Clean up after ourselves
    Release();
}
```

With the completion of these two functions, we have reached the conclusion of the API exposed by the compiler class as it will commonly be used by any front end application. We have covered each task including the importing of the scene, preparing that scene data for processing, creating and executing each compiler module process and finally saving the data back to file. What remains to be covered is of course each of the compiler process classes that perform the pre-process tasks available in this lab project.

Adding BSP Tree Support – CBSPTree.cpp

Before we can realistically move on to discussing the compiler module classes, we must first discuss the integration of the new polygon-aligned BSP compiler supported into this lab project. Although this class is a compilation module in its own right, much of the functionality contained within this class is also used by other processing modules. In the case of the hidden surface removal process – which is actually executed *before* the main BSP compile step – we must create and compile individual BSP tree objects from each of meshes that we intend to merge before we can perform the union operation. Since this class is a major pre-requisite in many cases, it will be beneficial to us if we step outside of the program flow for a while and examine the entire set of BSP tree classes and compilation procedures.

A Note Regarding Linked Lists

The various compiler classes that we have previously implemented have made extensive use of the STL 'list' template class to provide much of the linked list support required by those applications. In this new compiler application we use manually constructed linked lists almost exclusively. We have encountered these concepts many times in previous lessons, an example of which would be the sibling pointer and iteration logic used extensively in our coverage of the D3DXFRAME and D3DXMESHCONTAINER structures. We have chosen to use manual linked lists in this application due to the many varied situations in which they are used requiring a high degree of flexibility not easily implemented when using the STL list template class.

With that in mind, let us now move on to discuss each of the supporting classes required by the BSP tree compilation process.

The CBSPFace Class

Up until this point we have been using the renderable 'CFace' class during the manipulation of our scene data. This class maintains those pieces of information that our rendering application may require – such as the texturing, material and surface normal data imported from the source IWF file. Recall that the 'CFace' class is also derived from a base of 'CPolygon' that provides much of the vertex management we require such as the 'Split' function we encountered earlier. Due to the fact that we want to compile the same information that we will ultimately be rendering in our run-time application, this class is derived directly from the 'CFace' class. By extending the renderable polygon class in this way, we automatically inherit the variables and functionality exposed by this base class without having to explicitly redeclare such members.

The 'CBSPFace' class is used exclusively by the BSP tree compilation process, and contains several additional member variables that are designed specifically for those tasks undertaken by the BSP tree class.

```
class CBSPFace : public CFace
{
public:

    // Constructors & Destructors
    CBSPFace( );
    CBSPFace( const CFace * pFace );

    // Public Variables for This Class
    bool        UsedAsSplitter;
    long        OriginalIndex;
    bool        Deleted;
    long        ChildSplit[2];
    CBSPFace   *Next;
    long        Plane;

    // Public Virtual Functions for This Class
    virtual HRESULT Split( const CPlane3& Plane, CBSPFace * FrontSplit,
                           CBSPFace * BackSplit, bool bReturnNoSplit = false );

};
```

Let us briefly examine each of the new member variables declared within this extended polygon class.

bool UsedAsSplitter

In the previous lab project 16.1 in which we developed a BSP node tree compiler class, the polygons were removed from consideration and attached to each node at the point of being selected as a separating plane candidate. During the compilation of the leaf based BSP tree however, we are required to pass any surviving polygon fragments through the tree in order for them to be collected in the attached leaf structures that will be created at the terminal nodes. While this could be achieved by

passing the polygons through the tree after the nodes and leaves have been constructed, the most efficient approach is to perform the polygon splitting and classification at the same time as building the tree itself. To this end, we need to be able to determine if a polygon fragment has already been chosen as a node plane candidate in order to remove it from consideration as it is passed through the tree during construction.

The 'UsedAsSplitter' variable is the means by which this is achieved. By testing the value of this member in the splitter selection step, we can determine whether a polygon has already been used to generate a node at a point further up in the recursive building process. With a default value of 'false', each polygon passed into the BSP tree class for compilation can initially be chosen for node plane candidacy. Whenever a polygon has been used in the creation of a node however, this member variable should be set to 'true'. Thus, whenever that polygon is encountered by the splitter selection routine at a point further down the tree, it would not be selected for a second time. This leaves us free to pass as many polygons as we need through the recursive BSP compilation function, creating a leaf only when there are no remaining polygons available for use as splitters in the polygon list specified.

long OriginalIndex

When used in conjunction with the more traditional split type BSP tree compilation process, this variable is used to transfer the index of this polygon's final location within the BSP trees centralized polygon array for storage in the leaf structure. When the application requests that the BSP tree compiler build a structure in which the leaves reference the original unsplit versions of the polygon data however – the non-split resulting tree type – this variable is used to track the index of the original polygon from which this particular fragment was created. Whenever a polygon is split, this original index value is copied into each of the resulting fragments. This enables us to identify the polygon from the original imported data set when any polygon fragment reaches a terminal node. We will explore this topic in greater depth a little later.

bool Deleted

During the hidden surface removal process, there are situations in which a polygon may have been split into multiple fragments of which none have been removed under the clipping rules specified. We will discover shortly how it is possible for us to easily repair these unnecessary splits and restore the polygon to its initial state. To do this however, requires that we do not physically delete the original polygons whenever they are found to span a BSP tree node that subsequently caused it to be split. For this reason, we maintain this 'Deleted' variable that we use to notify the HSR clipping procedure that it should simply be ignored as if it had been removed altogether. This allows us to simply set this variable to 'false', and those of any resulting split fragments to 'true', in order to undo any unnecessary split operations that may have occurred.

long ChildSplit[2]

Used only by the hidden surface removal support functionality provided by the BSP tree class, each element in this array stores indices to the two resulting polygon fragments added to the resulting polygon list should this polygon be split against a node's separating plane. We will examine how this variable is applied when we come to discuss these supporting functions later in this chapter.

CBSPFace * Next

This pointer variable is used to describe the *next* 'CBSPFace' object contained within any individual linked list being maintained by the compiler. This variable is used in several situations, each of which will be explained when they are encountered.

long Plane

Earlier in this chapter we discussed how we might build an array of planes that could be referenced by the nodes of the tree using a simple index. This member variable contains an index that references an element within such a plane array that will later be constructed within the BSP tree class. Recall that using a combined plane array allows us to reduce the amount of memory and file resources required to store the plane information which might otherwise be duplicated for coplanar nodes and polygons. In addition to reducing the memory footprint of this plane data, we will also shortly discuss how the accuracy of the BSP compilation process can be greatly improved by building shared and reusable plane data in this way.

CBSPFace::CBSPFace()

There are two constructors provided by this class. The first of these is the default class constructor shown below.

```
CBSPFace::CBSPFace()
{
    // Initialise anything we need
    UsedAsSplitter  = false;
    OriginalIndex   = -1;
    Next            = NULL;
    Deleted         = false;
    ChildSplit[0]   = -1;
    ChildSplit[1]   = -1;
    Plane           = -1;
}
```

As you can see, this constructor simply initializes each of the extended class member variables with a sensible default value. The only member whose default value is of any real importance here is the 'UsedAsSplitter' variable. Due to the fact that every polygon used in the construction of the BSP tree should be used as a node plane candidate at some point in the process, it is imperative that this variable is defaulted to 'false'. Remember that in C++, each of the base class constructors will be automatically executed in order. As a result, it is not necessary for us to default those member variables declared by the parent 'CFace' or 'CPolygon' classes here.

The second constructor defined by 'CBSPFace' class is an overload that accepts a single parameter of a 'CFace' object pointer type. This constructor is used in cases where the data contained within the polygon passed to this constructor should be duplicated into the new 'CBSPFace' polygon being constructed. This function begins in a similar manner to that of the default constructor by initializing the extended member variables declared by this class.

```
CBSPFace::CBSPFace( const CFace * pFace )
{
    // Initialise anything we need
    UsedAsSplitter  = false;
```

```
    OriginalIndex   = -1;
    Next            = NULL;
    Deleted         = false;
    ChildSplit[0]   = -1;
    ChildSplit[1]   = -1;
    Plane           = -1;
```

Once the ‘CBSPFace’ member variables have been set to their initial state, this function then proceeds to duplicate the information contained within the ‘CFace’ object passed to this constructor. As demonstrated below, this is achieved by first copying the values from those member variables declared by the ‘CFace’ class from the polygon object passed, into the corresponding members inherited by this object. Once this information has been transferred, the constructor then proceeds to duplicate the vertex data contained within the specified polygon object into the internal vertex array of the new polygon object being constructed.

```
    // Duplicate required values
    Normal          = pFace->Normal;
    TextureIndex    = pFace->TextureIndex;
    MaterialIndex   = pFace->MaterialIndex;
    ShaderIndex     = pFace->ShaderIndex;
    Flags           = pFace->Flags;
    SrcBlendMode    = pFace->SrcBlendMode;
    DestBlendMode   = pFace->DestBlendMode;

    // Duplicate Arrays
    AddVertices( pFace->VertexCount );
    memcpy( Vertices, pFace->Vertices, VertexCount * sizeof(CVertex));
}
```

CBSPFace::Split

Just as with the ‘CFace' class from which ‘CBSPFace’ is derived, we must provide an overloaded ‘Split’ function that is responsible for copying any additional member variables maintained by this class into each of the two split fragments. Notice that at the very start of this function we make a call to the base class’ implementation of the ‘Split’ procedure. Recall that in this case, the base from which this polygon class is derived is ‘CFace’ which is in turn derived from ‘CPolygon’. By calling the ‘CFace’ implementation of the split function, which in itself also calls the ‘CPolygon’ implementation, we ensure that all of those pieces of information maintained by either class in the hierarchy are duplicated into the two resulting polygon fragments.

```
HRESULT CBSPFace::Split( const CPlane3& SplitPlane, CBSPFace * FrontSplit,
                         CBSPFace * BackSplit )
{

    // Call base class implementation
    HRESULT ErrCode = CFace::Split( SplitPlane, FrontSplit, BackSplit );
    if (FAILED(ErrCode)) return ErrCode;

    // Copy remaining values
    if (FrontSplit)
    {
```

```
        FrontSplit->UsedAsSplitter = UsedAsSplitter;
        FrontSplit->OriginalIndex  = OriginalIndex;
        FrontSplit->Plane          = Plane;

    } // End If

    if (BackSplit)
    {
        BackSplit->UsedAsSplitter = UsedAsSplitter;
        BackSplit->OriginalIndex  = OriginalIndex;
        BackSplit->Plane          = Plane;

    } // End If

    // Success
    return BC_OK;
}
```

In this function, we copy three main pieces of information into each of the resulting polygon fragments. The first of these is the 'UsedAsSplitter' member. It is important that each of the split fragments inherit the value stored in this member whenever a polygon is split during the BSP compilation process. If this were not the case then, whenever a polygon that has already been selected for node plane candidacy is split, each child fragment would subsequently cause a new node plane to be generated at some point underneath the node originally generated. While this would not necessarily harm the integrity of the BSP tree or the validity of the solid / empty leaf information, creating these additional nodes would be wasteful and would merely increase the time it would take to traverse the hierarchy.

Secondly, we copy the source polygon's 'OriginalIndex' member value into each fragment. Again this is only really important when the application has chosen to construct a non-split resulting BSP tree, but we copy this information regardless of the tree type. As mentioned earlier, this index is used as a reference to the original scene polygon that was initially used to create each 'CBSPFace' object used in the BSP compilation process. Whenever a polygon gets split into two new fragments, each of these resulting polygons should maintain that same 'OriginalIndex' value so that we can correctly process this information when we reach a terminal node.

Finally, we duplicate the plane index into each of the resulting front and back fragments. Since each of the fragments generated by the polygon splitting operation are absolutely **guaranteed** to be coplanar with the source polygon, both of these fragments can share the same plane structure referenced by this polygon.

The CBSPLeaf Class

In the previous implementations of our kD-tree, oct-tree and quad-tree classes, we made extensive use of the 'CBaseLeaf' class to attach various pieces of information to each terminal node. This information included items such as the axis aligned bounding box of the space contained within the leaf, in addition to storing references to any polygon fragments that were contained within that leaf node's interior. The leaf structure used by the new BSP tree compiler class bears a striking resemblance to that original leaf concept.

```
class CBSPLeaf
{
public:

    // Constructors & Destructors
             CBSPLeaf( );
    virtual ~CBSPLeaf( );

    // Public Variables for This Class
    std::vector<long>   FaceIndices;
    CBounds3            Bounds;

    // Public Member Functions Omitted
};
```

Although our previous implementation of the leaf class contained many more run-time specific member variables, the basic premise is the same. This class stores a list of every polygon that was collected at any terminal node by the hierarchy compilation process, in addition to the axis aligned bounding box information that has been so useful to us in the past.

Let us examine each of these member variables before we move on to discuss the methods exposed by this class.

std::vector<long> FaceIndices

Unlike the previous implementation of the 'CBaseLeaf' class in which we stored direct pointers to scene polygon objects, recall that we are using an index based, file friendly mechanism in order to reference the different pieces of information in the tree. The 'FaceIndices' member variable therefore, stores a list of indices that reference individual elements with the BSP tree class's internal polygon array. Because this array need only store simple integer values that do not change after the leaf has been built, this member uses a simple STL vector type to provide an easy means by which to insert and access the polygon index data. Once the tree has been built, the indices stored in each leaf can then be used to retrieve the relevant data from the BSP tree's centralized polygon array.

CBounds3 Bounds

The 'Bounds' member is of the type 'CBounds3' – one of our new math utility classes – and is used to store the world space axis aligned extents of every polygon collected and stored within this leaf. This information will be written out to the export file along with the leaf structure, and can be imported and used in a run-time / rendering application to provide additional frustum culling or broad phase information when collect leaves for further consideration.

CBSPLeaf::BuildFaceIndices

The 'BuildFaceIndices' method is the only function defined within the 'CBSPLeaf' class. This function is designed to populate the polygon index data contained within the 'FaceIndices' STL vector. This function is called only by the BSP compiler's recursive 'BuildBSPTree' function and accepts a single parameter that describes the first item in a linked list of polygons to be stored within this leaf. Prior to calling this function, the 'BuildBSPTree' function updates the 'OriginalIndex' member of each 'CBSPFace' object contained within this list. Each 'OriginalIndex' variable is assigned a value that

describes the final location within the BSP tree's internal polygon array into which the polygon has been inserted. This index information is subsequently used to populate the member 'FaceIndices' array within this function.

The first thing we do in this function is to reset the bounding box member that will be used to describe the world space extents of every polygon stored within this leaf. This bounding box will be constructed incrementally as each polygon within the specified linked list is visited later in this function.

```
bool CBSPLeaf::BuildFaceIndices( CBSPFace * pFaceList )
{

    CBSPFace       *Iterator      = pFaceList;
    unsigned long   OriginalIndex = 0;

    // Reset bounds, will be rebuilt
    Bounds.Reset();
```

With the 'Bounds' member reset, we can now begin to iterate through the specified polygon list. Notice in the previous code snippet how the pointer describing the first element of the list is copied into a local variable named 'Iterator'. This variable will be used as the means by which we iterate through the linked polygon list.

While traversing the linked list of polygon data passed to this function, the first thing we must do is extract the 'OriginalIndex' value from the current polygon that will be added to the 'FaceIndices' member array. Before this happens however, this function includes a test to determine whether or not this index has already been added to the leaf's internal polygon list. This is achieved with a call to the 'std::find' library function that is designed to iterate through an STL vector, searching for an element that contains the specified value. In doing so, we prevent duplicate polygon indices from being added to the same leaf for whatever reason.

If no duplicate was found in the 'FaceIndices', we are now free to add the polygon index to this array. Due to the fact that we are adding many potential polygon indices to the STL vector member within a loop however, we use the capacity threshold resizing logic in order to prevent this array from being resized / reallocated for every index added. We have observed this kind of vector resizing logic in the past, but to summarize the process used here; if the total number of elements in the vector *including* the new index is found to exceed its current internal capacity then we allocate additional space by calling the vector's 'Reserve' method. To this method we specify a capacity equal to the current number of items stored in the vector plus an additional number of slack elements into which the vector can grow. In this way, the vector will only ever be physically resized each time we have added a number of indices equal to this additional slack space. The number of additional elements reserved in this function is defined by the constant shown below.

```
#define BSP_ARRAY_THRESHOLD 100
```

Once we have reserved any required number of elements in the 'FaceIndices' member, we can then add the 'OriginalIndex' value taken from the current polygon with a call to the STL vector's 'push_back' method.

```
    // Iterate building list
    while ( Iterator != NULL )
    {
        // Get the original polygon's index
        OriginalIndex = Iterator->OriginalIndex;

        try
        {
            // Make sure this index does not already exist in the array?
            if ( std::find(FaceIndices.begin(), FaceIndices.end(),
                           OriginalIndex) == FaceIndices.end())
            {

                // Resize the vector if we need to
                if (FaceIndices.size() >= (FaceIndices.capacity() - 1))
                {
                    FaceIndices.reserve( FaceIndices.size() +
                                         BSP_ARRAY_THRESHOLD );
                } // End If

                // Finally add this face index to the list
                FaceIndices.push_back(OriginalIndex);

            } // End If needs adding

        } // End Try Block
```

If an exception occurred during any of the above operations, we need to catch this situation and release any resources that were allocated before returning a failure code to the calling function

```
        catch (...)
        {
            // Clean up and bail
            FaceIndices.clear();
            return false;
        } // End Catch
```

In the final stages of this loop we must now grow the leaf's bounding box to include the current polygon's vertex data. This is done with a call to the 'CalculateFromPolygon' method of the leaf' 'Bounds' member. Recall that, in order for this function to grow the bounding box extent values using the specified vertex data, we must pass a value of false to the final 'reset' parameter.

With this polygon fully considered, we finally move on to the next polygon ready for the following iteration of the while loop. If this polygon was the last item in the linked list, the 'Next' variable will contain a value of NULL that would be subsequently transferred into the local 'Iterator' pointer, causing the while loop to exit.

```
        // Build into current bounding box
        Bounds.CalculateFromPolygon( Iterator->Vertices, Iterator->VertexCount,
                                     sizeof(CVertex), false );
```

```
        // Move to next poly
        Iterator = Iterator->Next;

    } // End While
```

At this stage, the 'FaceIndices' STL vector will contain an index to each polygon passed in to this leaf for storage. When adding each of the polygon indices to this array, recall that we used a capacity threshold technique to prevent the STL vector from having to be reallocated and duplicated every time a new index value was added. As we know, this block resizing technique can result in an STL vector having an internal capacity that is much larger than might actually be required to store the index values. In order to prevent these arrays from needlessly consuming this additional slack memory, the final task undertaken by this function is to optimize the memory allocated by the 'FaceIndices' vector if required.

```
    // Optimize the vector
    if (FaceIndices.size() < FaceIndices.capacity())
        FaceIndices.resize( FaceIndices.size() );

    // Success
    return true;
}
```

The CBSPNode Class

This class is used to represent the individual nodes within the hierarchy of the new BSP leaf tree. Similar to the node class we have previously implemented in both the kD-tree and BSP node tree compiler projects, this node class is represented by a single separating plane used to partition the space on either side into two equal halfspaces. This is a relatively simple class that is based on those we are already familiar with so let us move straight on to examining the class declaration.

```
class CBSPNode
{
public:

    // Constructors & Destructors
             CBSPNode( );
    virtual ~CBSPNode( ) {};

    // Public Variables for This Class
    long            Plane;
    long            Front;
    long            Back;
    CBounds3        Bounds;

    // Public Member Functions Omitted
};
```

At this point, it should be reasonably clear as to the purpose of each of the member variables declared within this class. In the interest of completeness however, let us just briefly examine each member before moving on to discuss the class methods.

long Plane

Just as with the 'CBSPFace' class, the node maintains an index into the BSP tree's internal array in order to describe the plane on which this node lies. This index is taken directly from the polygon object that was used as the candidate plane for the generation of this node. In this way we are able to reuse not only that plane information shared between individual polygons, but also between coplanar nodes and polygons alike.

long Front

This member variable has a dual purpose. Recall from our earlier discussion of the file based BSP data structures; we will use the same front member variable to attach both nodes *and* leaves to the appropriate side of this node. The value stored in this member is used to represent an index into one of the BSP tree's centralized data arrays. The type of data structure that is attached to the front of this node via this member depends on the sign of the value stored here. If this variable contains a negative value, this means that there is a leaf attached to the front of the node and is used to represent an index to an element within the BSP tree's leaf array. If this value is positive, then there is simply another node attached as the front child. This is then used as the index into the BSP tree's node array.

long Back

This purpose and use of this member variable is identical to that of the aforementioned 'Front' member. The only difference here is that this member is used to describe the index of either the node or leaf attached as the child *behind* this node.

CBounds3 Bounds

This member represents the axis aligned world space extents for this node. Recall that the bounding box contained within a BSP tree node is used to describe the world space extents of *every* polygon that will exist in the tree at some point beneath it in the hierarchy.

CBSPNode::CBSPNode()

This class contains only a single default constructor in which each of the primitive typed member variables are defaulted to a value of '-1'. This value is used to signify that the information has not yet been initialized. The only variable that we do not initialize here is the 'Bounds' member. This is due to the fact that the constructor of this object will be called implicitly, whenever an instance of this node class is created.

```
CBSPNode::CBSPNode()
{
    // Initialise anything we need
    Plane   = -1;
    Front   = -1;
    Back    = -1;
}
```

CBSPNode::CalculateBounds

The only member function defined within this class is the 'CaclulateBounds' method shown in the following block of code. Called during the BSP construction process, this is a utility function designed

to populate the ‘Bounds’ member of this object with the world space extents of every polygon that will exist in the tree at some point underneath this node.

This method accepts two parameters. The first of these two describes the first element in a linked list of ‘CBSPFace’ object pointers that will be taken into consideration during the calculation of this bounding box. The second parameter is a simple Boolean value that dictates whether the specified polygon data should be used to simply grow any existing bounding box values, or whether it should first be reset.

```
void CBSPNode::CalculateBounds( CBSPFace * pFaceList, bool ResetBounds )
{
    CBSPFace *Iterator = pFaceList;

    // Reset bounding box and grow from default state?
    if ( ResetBounds ) Bounds.Reset();

    // Calculate bounding box based on the face list passed
    while ( Iterator )
    {
        // Grow the bounding box values
        Bounds.CalculateFromPolygon( Iterator->Vertices, Iterator->VertexCount,
                                     sizeof(CVertex), false );

        // Move on to the next polygon
        Iterator = Iterator->Next;

    } // End If
}
```

The techniques used in the development of this function have been discussed in detail elsewhere in this lesson, so we will not go into any further detail here. We will discuss how and when this function is to be used shortly, as we continue on to discuss the main leaf based BSP tree compilation class in the following section.

The CBSPTree Class

Up until this point we have spent a great deal of time focusing on the specifics of our new pre-processing tool and the various associated framework classes that have been introduced. These discussions were vital in order for us to understand the techniques involved in building the compiler application for lab project 16.2. In addition, this coverage has laid the foundation that will allow us to discuss the various compilation modules, without having to stop and examine any new support and utility functionality utilized in the process. Let us therefore continue on to examine the new polygon aligned BSP leaf tree compiler that we have made reference to several times in this chapter.

While the ‘CBSPTree’ class serves as the basis for many of the compilation modules that are covered in both this lesson and the next, it also acts as a compilation module in its own right. Referring back to our discussion of the ‘PerformBSP’ method of the ‘CCompiler class, we know that a BSP tree is constructed from the scene information in much the same way as we have done in each of our previous spatial hierarchy lab project applications. This has traditionally involved passing scene polygon data into the compiler class, triggering the compilation process and finally interpreting and storing the compiled

information in a manner suitable for rendering. In this pre-processing application, the high level concepts involved in the use of the BSP tree class as a compilation module are very similar. The one exception is that, after compiling the tree we must interpret and store the compiled information in a manner suitable for export. With this in mind, let us now examine the declaration of the new 'CBSPTree' class in order to see just what information will be built.

As before, we have removed the list of member functions in this declaration to improve readability, so check the source code for a full list of methods.

```
class CBSPTree
{
public:

    // Constructors & Destructors for This Class.
             CBSPTree();
    virtual ~CBSPTree();

private:

    // Private Variables for This Class.
    CBSPFace       *m_pFaceList;
    unsigned long   m_lActiveFaces;
    vectorNode      m_vpNodes;
    vectorPlane     m_vpPlanes;
    vectorLeaf      m_vpLeaves;
    vectorBSPFace   m_vpFaces;
    vectorBSPFace   m_vpGarbage;
    CBounds3        m_Bounds;
    BSPOPTIONS      m_OptionSet;
    ILogger        *m_pLogger;
    CCompiler      *m_pParent;
};
```

As you can see, there are many variables declared within this new 'CBSPTree' class. While we are familiar with the types of information stored within many of these members at this point, there are several items in this list that we have not yet encountered. Therefore, before we move on to the discussion of the class implementation, let us first familiarize ourselves with the role of each of these member variables.

CBSPFace * m_pFaceList

Before we can even begin to think about compiling the tree structure, we obviously must have access to the list of polygons to which the nodes of the polygon-aligned BSP tree will be oriented. This member variable contains the first / head element within a linked list of 'CBSPFace' objects that will be used as the source set for the BSP compilation process. In the earlier discussion of the 'CCompiler::PerformBSP' function, we saw how each polygon in the scene was passed into the BSP tree object through a call to its 'AddFaces' method. This function accepted an array of 'CFace' object pointers that were to be considered as valid polygons for use by the BSP tree compilation process. We will discuss how the 'AddFaces' method is implemented shortly, but in summary this function takes each of the specified 'CFace' objects, duplicates that polygon's data into a new 'CBSPFace' object before finally attaching that new polygon to the linked list maintained by this member.

vectorNode m_vpNodes

The ‘m_vpNodes’ member is used as the centralized container array for the *final* list of hierarchy nodes contained within the compiled BSP tree. The information contained within this node array will later be exported to file after each compilation process has been run. This member variable is of the type ‘vectorNode’ which is a typedef of a standard STL vector template designed to store a series of ‘CBSPNode’ object pointers.

```
typedef std::vector<CBSPNode*>   vectorNode;
```

In order to access the data stored within this vector easily and safely, the BSP tree class defines both the ‘GetNode’ and ‘GetNodeCount’ accessor functions for use by the application. Existing elements can also be assigned with the use of the ‘SetNode’ method. A list of these accessor functions can be found and the end of this variable listing if you wish to examine them.

vectorPlane m_vpPlanes

This member variable is intended to store a list of every plane used by both the polygons and nodes contained within this BSP tree. Recall that we will be relying heavily upon shared plane information in this BSP tree implementation in order to save space and – as we will discover shortly – to improve the accuracy of the BSP compilation process. As a result, individual planes stored within this array may be referenced by many different sources before, after and during the compilation process. The information contained within this plane array will also later be exported to file after each compilation process has been run.

The ‘m_vpPlanes’ member is of the type ‘vectorPlane’ which is a typedef of a standard STL vector template designed to store a series of ‘CPlane3’ object pointers.

```
typedef std::vector<CPlane3*>    vectorPlane;
```

In order to access the data stored within this vector easily and safely, the BSP tree class defines both the ‘GetPlane’ and ‘GetPlaneCount’ accessor functions for use by the application. Existing elements can also be assigned with the use of the ‘SetPlane’ method.

vectorLeaf m_vpLeaves

The ‘m_vpLeaves’ member is used as the centralized container array for the *final* list of leaf objects contained within the compiled BSP tree. The information contained within this leaf array will later be exported to file after each compilation process has been run. This member variable is of the type ‘vectorLeaf’ which is a typedef of a standard STL vector template designed to store a series of ‘CBSPLeaf’ object pointers.

```
typedef std::vector<CBSPLeaf*>   vectorLeaf;
```

In order to access the data stored within this vector easily and safely, the BSP tree class defines both the ‘GetLeaf’ and ‘GetLeafCount’ accessor functions for use by the application. Existing elements can also be assigned with the use of the ‘SetLeaf’ method.

vectorBSPFace m_vpFaces
The 'm_vpFaces' member is used as the centralized container array for the *final* list of polygon objects contained within the compiled BSP tree. The information contained within this polygon array will later be exported to file after each compilation process has been run. This member variable is of the type 'vectorBSPFace' which is a typedef of a standard STL vector template designed to store a series of 'CBSPFace' object pointers.

```
typedef std::vector<CBSPFace*>   vectorBSPFace;
```

In order to access the data stored within this vector easily and safely, the BSP tree class defines both the 'GetFace' and 'GetFaceCount' accessor functions for use by the application. Existing elements can also be assigned with the use of the 'SetFace' method.

vectorBSPFace m_vpGarbage
Logically, as the tree compilation process becomes ever more comprehensive, there are likewise more situations in which failures might occur. If a failure does happen, it is always preferable for the application to gracefully handle this situation such that the user might take further action to resolve the source of the problem. This member array is provided in order to make the process of handling such errors a little easier during the recursive BSP compilation procedure. By providing a convenient location for the collection of polygons that should be released at a later point in time, we can actually bypass some significant problems introduced when trying to clean up any allocated polygon data in the middle of this heavily recursive process. The specifics of the error handling and garbage collection scheme will be covered a little later on in this section.

CBounds3 m_Bounds
As with many of the classes to which we have previously been introduced, the BSP tree class also stores an axis aligned bounding box that is used to represent the total world space extents of every node, leaf and polygon contained within the compiled tree. This information will be crucial during the portal compilation process we will develop in the next lesson.

unsigned long m_lActiveFaces
This member is used primarily for the purposes of informing the user about the progress of the compilation process. The value stored here simply maintains a count of the number of polygons currently active within the compilation process. With this information we can determine how far through the BSP compile process we have come simply by comparing this total number of polygons against the number of polygons that have been removed from consideration at each step.

BSPOPTIONS m_OptionSet
This member stores the various settings, specified by the application to inform the compiler how the BSP tree should be compiled. This options structure is set using the 'SetOptions' accessor function outlined on the following page.

ILogger * m_pLogger
In order for the BSP compiler to report progress, error and useful status information to the user, this member variable stores a pointer to the application defined logging class. While the logging feature remains optional, this member can be set with a call to the 'SetLogger' accessor function defined by this

class. In this lab project, the logging class instance is passed to this object by the 'CCompiler::PerformBSP' function.

CCompiler * m_pParent

Each compilation module class maintains a member that stores a reference to the parent compiler class that created it. This member is set with a call to the 'SetParent' accessor function defined by each of these classes. This information is used by each of these processing modules in order to test the current state of the compiler to determine if the compile operation has been either paused or cancelled. Although we have omitted much of the logging and progress functionality from the code listings in this workbook, you should take a look at the source code for lab project 16.2 to see this process in action.

BSP Tree Accessor Functions

There are many accessor functions defined within the BSP tree class. In each case, the body for these functions can be found in the class declaration contained in the header file, rather than in the source module / .CPP file. We include the function bodies in the header mostly for convenience. However, because these functions are very simple and are accessible by any source module that might include this header, these functions are much more likely to be chosen to be inline functions. Due to the fact that these methods are so small, and will be called often, this is an ideal situation.

We will not go into detail about how each of these functions is implemented, for this you should take a look at the 'CBSPTree.h' header file contained in the 'Compiler Source' project directory. Instead we will give a brief overview of each category of accessor function available in this class.

The first series of accessor functions defined within the BSP tree class are those which return the number of elements stored within the STL vector for each type of data stored. These functions wrap a call to the STL vector 'size' method, and simply return that information to the calling function.

```
unsigned long   GetNodeCount  ( ) const
unsigned long   GetLeafCount  ( ) const
unsigned long   GetPlaneCount ( ) const
unsigned long   GetFaceCount  ( ) const
```

The second set of accessor functions defined here are those that actually retrieve a pointer to each specific data class from the member arrays. Each of these functions accepts a single parameter that is used to specify the index to the element in the array which the calling function would like to retrieve. Rather than simply have the underlying STL vector throw an exception if the specified index is out of bounds however, these functions test the value of the index parameter, returning NULL if it exceeded the end of the array. This allows the calling function to retrieve each type of object safely without having to wrap each call with exception handling logic.

```
CBSPNode        *GetNode       ( unsigned long Index ) const
CPlane3         *GetPlane      ( unsigned long Index ) const
CBSPLeaf        *GetLeaf       ( unsigned long Index ) const
CBSPFace        *GetFace       ( unsigned long Index ) const
```

The third set of accessor functions are called in order to *set* an individual element in each of the member arrays. These functions each accept two parameters. The first is the index to the element in the relevant

array that the calling function would like to set. The second is a pointer to the data object that should be stored in that specified array element.

```
void            SetNode       ( unsigned long Index, CBSPNode   * pNode  )
void            SetPlane      ( unsigned long Index, CPlane3    * pPlane )
void            SetLeaf       ( unsigned long Index, CBSPLeaf   * pLeaf  )
void            SetFace       ( unsigned long Index, CBSPFace   * pFace  )
```

The fourth and final set of accessor functions are those that set the various pieces of information required by each compilation process module. These include the process options, a pointer to the parent compiler and a pointer to the class defined logging class. These functions simply store those specified values into the applicable member variables declared by this class.

```
void            SetOptions    ( BSPOPTIONS Options    )
void            SetParent     ( CCompiler * pCompiler )
void            SetLogger     ( ILogger * pLogger     )
```

Now that we have familiarized ourselves with each component of the 'CBSPTree' class declaration, let us move directly on to the implementation of the all important class methods.

CBSPTree::AddFaces

The 'AddFaces' method is the means by which any applicable scene polygon data is added to the tree ready for compilation. We have seen how this method is called in our earlier coverage of the 'CCompiler::PerformBSP' function.

```
HRESULT CBSPTree::AddFaces( CFace ** ppFaces, unsigned long lFaceCount )
{
    CBSPFace *NewFace = NULL;

    // Validate Parameters
    if ( !ppFaces || lFaceCount <= 0 ) return BCERR_INVALIDPARAMS;

    // Release the old tree data (if any)
    if (!m_pFaceList) ReleaseTree();
```

This function accepts two parameters. The first is an array of 'CFace' object *pointers* that contains the list of polygons that have been selected by the application to be included in the BSP compilation process. The second parameter specifies the number of polygons contained within that array.

After ensuring that each of the specified parameters are valid, this function first releases any tree data that already exists within this object should the class have been previously used. This is only done during the first call to the 'AddFaces' function however. We can determine if this is the first time that this function has been called by testing the value stored within the 'm_pFaceList' member variable. If this variable currently contains a NULL pointer, this means that no polygon data has yet been collected. This is something that will no longer be the case after this function has completed its first execution.

At this stage, we begin to loop through each of the polygons passed into this function ready to populate the 'm_pFaceList' linked list member variable. This list will contain the data that is actually used to

construct the BSP tree in the next step. To this end, each valid ‘CFace' polygon object encountered in the array passed to this function is duplicated into a newly allocated ‘CBSPFace’ polygon with a call to the ‘AllocBSPFace’ function. This function wraps much of the allocation and error handling logic on our behalf and is an easy and safe way to create a new BSP specific polygon object. Into this newly duplicated polygon we store the index that describes the order in which this polygon was created. This value is retrieved from the ‘m_lActiveFaces’ member variable which is subsequently incremented.

```
    // First add the mesh polygons to the initial linked list
    for ( unsigned long i = 0; i < lFaceCount; i++ )
    {

        // Add if available
        if (ppFaces[i])
        {
            // Allocate the new BSPFace
            if (!(NewFace = AllocBSPFace( ppFaces[i] )))
            {
                ReleaseTree();
                return BCERR_OUTOFMEMORY;
            } // End If Out of Memory

            // Store original index for later use
            NewFace->OriginalIndex = m_lActiveFaces;
            m_lActiveFaces++;
```

With this source polygon duplicated into the new ‘CBSPFace’ typed object, we attach this new polygon to the head of the internal source polygon linked list referenced by the ‘m_pFaceList’ member.

```
            // Attach this to the list
            NewFace->Next = m_pFaceList;
            m_pFaceList   = NewFace;
```

In the spatial hierarchy concepts developed in previous lab projects, we had the ability to generate trees in which the polygon data stored in its leaves were not split against the separating planes. When the non-split option was selected in these earlier compiler classes, the source polygon data used in the construction of the tree remained in the tree’s internal polygon array from the start. This was in contrast to the splitting method in which the polygon fragments were added to the internal polygon array only when they were being added to a leaf in the tree. This same logic is applied to our new BSP leaf tree compiler.

In the following block of code we can see that the current original source polygon is duplicated and added directly to the tree’s final polygon array if the non-split option was selected by the application. Recall that earlier in this function we stored the current face count in the ‘OriginalIndex’ member of each polygon stored in the linked list. If the non-split resulting tree option has been selected, this ‘OriginalIndex’ value will correspond with the element in which these duplicate polygons have been stored in the tree’s final polygon array. We will examine how all of this is tied together in the coverage of the ‘ProcessLeafFaces’ method later in this section.

```
            // Non Split tree?
            if ( m_OptionSet.TreeType == BSP_TYPE_NONSPLIT )
```

```
            {
                // Allocate some storage space
                if (!IncreaseFaceCount())
                {
                    ReleaseTree();
                    return BCERR_OUTOFMEMORY;
                } // End if out of memory

                // Allocate a duplicate of the original original
                if (!(NewFace = AllocBSPFace( ppFaces[i] )))
                {
                    ReleaseTree();
                    return BCERR_OUTOFMEMORY;
                } // End if out of memory

                // Store this new pointer
                SetFace( GetFaceCount() - 1, NewFace );

            } // End if backup required

        } // End If Face Available
```

With the current polygon now processed, we move onto the next item in the specified array. Once the loop has run its course, this function returns a success code back to the calling function.

```
    } // Next Face

    // Success
    return BC_OK;
}
```

CBSPTree::CompileTree

After having added each of the polygons to be considered by the BSP tree compiler, this function is called in order to begin the actual tree compilation process. Put simply, this method wraps each of the calls to the various pieces of functionality required in order to construct the BSP tree.

The first process that we must undertake when building the BSP tree in this application is to construct the combined plane array into which each ‘CBSPFace’ object’s ‘Plane’ member will reference. This information is used both to reduce the amount of memory consumed for plane data between coplanar polygons, in addition to improving the accuracy of the BSP compilation process as we shall demonstrate shortly.

Once the pre-compiled plane array information has been generated, we are ready to begin the construction of the BSP tree itself. Before we can do this however, we must first allocate the root node that will be passed into the initial call of the main recursive build procedure. This is done with a call to the ‘IncreaseNodeCount’ method discussed later in this section.

```
HRESULT CBSPTree::CompileTree( )
{
    HRESULT ErrCode;
```

```
    // Validate values
    if (!m_pFaceList) return BCERR_INVALIDPARAMS;

    // Calculate our plane array
    if (FAILED(ErrCode = BuildPlaneArray())) return ErrCode;

    // Allocate our root node
    if (!IncreaseNodeCount()) return BCERR_OUTOFMEMORY;

    // Log – BSP: Building BSP Tree
```

At this point, we make a call to the 'BuildBSPTree' function, passing in both the index to the root node we allocated a moment before (the index of the root node will always be zero in this implementation), in addition to the linked list of polygon data referenced by the 'm_pFaceList' member.

```
    // Compile the BSP Tree
    if (FAILED(ErrCode = BuildBSPTree(0, m_pFaceList))) return ErrCode;
```

Assuming no errors occurred during the compilation process above, we should now have compiled a fully functional BSP tree. With each of the polygons in the source linked list now stored within the BSP tree data structures, we no longer own these polygons. As a result, we reset the 'm_pFaceList' with a NULL pointer value to remove all references to those polygons should we attempt to release this list, or append new polygons to it with future calls to 'AddFaces'.

```
    // Reset Face List
    m_pFaceList = NULL;
```

Now that we have populated the internal polygon array with the final set used in the construction of the BSP tree, the final task undertaken by this function is to calculate the tree's overall bounding box. We can do this quickly and efficiently using the 'CalculateFromPolygon' method exposed by the 'm_Bounds' object, passing in the vertex data from each polygon now stored in the tree as a whole.

With this final piece of information built, we finally return back to the calling function, notifying it of our success.

```
    // Build the trees bounding box
    m_Bounds.Reset();
    for ( ULONG i = 0; i < GetFaceCount(); i++ )
    {
        m_Bounds.CalculateFromPolygon( GetFace(i)->Vertices,
                                       GetFace(i)->VertexCount,
                                       sizeof(CVertex), false );

    } // Next Face

    // Log – BSP: Compile Success

    // Success
    return BC_OK;
}
```

CBSPTree::BuildPlaneArray

We have discussed at several points within this lesson just how beneficial the construction of a combined plane array can be during the compilation of a polygon aligned BSP tree. Not only does this pre-building technique provide a much smaller memory and file footprint with respect to the plane data itself, it also allows us to construct a BSP compiler class that is extremely stable and as fault tolerant as possible. Regarding the latter, we will observe later how building this plane array in advance can eliminate hundreds of potentially error prone polygon and vertex classification tests to be replaced with a simple integer equality test. Before we get to that point however, let us first examine how this new combined plane array is constructed. This should give us a better understanding of how these methods employ this information later in this chapter.

```
HRESULT CBSPTree::BuildPlaneArray()
{
    CBSPFace * Iterator = NULL;
    CPlane3    Plane, * TestPlane;
    CVector3   Normal, CentrePoint;
    int        i, PlaneCount;
    float      Distance;

    // Log – BSP: Building Initial Plane Array
```

This function accepts no parameters and employs a simple HRESULT style return code system for reporting any errors that may have occurred.

Given the list of source polygons contained within the ‘m_pFaceList’ member, this function begins by looping through each of these polygons using the standard linked list iteration logic we should already be familiar with. For each polygon in the list, the first job is to construct a ‘CPlane3’ object that describes the plane on which this polygon lies. As we know, in order to construct a plane from a polygon we need two pieces of information. The first is the polygon normal and the second is a point that lies on the intended plane. For the point we might perhaps use the position of the first vertex stored within the polygon vertex array. Due to the limited accuracy of single precision floating point values however, there is no guarantee that every vertex in the polygon lies on *exactly* the same plane. In many cases there will be a very slight twist to the polygon due to the vertices becoming slightly misaligned. As a result, in such a accuracy sensitive situation, it is not really ideal that we select any one vertex as the point used to construct the plane. For this reason, the first thing we do inside the main loop is to sum the position of every vertex in the current polygon, and then divide by the total number of vertices. This will give us an average center point that should provide us with a more accurate point on which to base the plane.

With the surface normal taken directly from the polygon and the averaged center point now computed we construct a new plane object by passing the normal and point information to the relevant overloaded ‘CPlane3’ constructor. This will automatically convert the normal/point information into the normal/distance values maintained by the plane class.

```
    // Loop through each face
    for ( Iterator = m_pFaceList; Iterator; Iterator = Iterator->Next )
```

```
    {
        // Calculate polygons centre point
        CentrePoint = Iterator->Vertices[0];
        for ( i = 1; i < (signed)Iterator->VertexCount; i++ )
              CentrePoint += Iterator->Vertices[i];

        // Average Vertices
        CentrePoint /= (float)Iterator->VertexCount;

        // Calculate polygons plane
        Plane = CPlane3( Iterator->Normal, CentrePoint );
```

Once we have generated the plane based on the information contained within the current polygon, this function then proceeds to check if a similar plane already exists in the plane array constructed so far. We can do this simply by comparing both the normal and the distance values contained within each plane. If both planes are equal – remembering to use a tolerance in order to ensure that we cater for any floating point precision errors – then we simply break from the loop having found a matching plane.

```
        // Search through plane list to see if one already exists
        PlaneCount = GetPlaneCount();
        for ( i = 0; i < PlaneCount; i++ )
        {
            // Retrieve the plane details
            TestPlane = GetPlane(i);

            // Test the plane details
            if ( fabsf(TestPlane->Distance - Plane.Distance) < 1e-5f &&
                 Plane.Normal.FuzzyCompare( TestPlane->Normal, 1e-5f )) break;

        } // Next Plane
```

If the previous loop ran to its conclusion then we know that no plane matching that of the current polygon was found in the plane array. We can test for this case by checking to see if the loop counter variable ‘i' is equal to the number of planes that were to be tested in that loop – ‘PlaneCount’. If this is the case then this function proceeds to add the unique plane described by this polygon the plane array stored within this class.

Now that we have an index to either an already existing plane, or the newly generated plane, this index must finally be stored in the polygon’s ‘Plane’ member ready for use in the BSP compilation process.

```
        // Add plane if none found
        if ( i == PlaneCount )
        {
            if (!IncreasePlaneCount()) return BCERR_OUTOFMEMORY;
            *GetPlane( PlaneCount ) = Plane;

        } // End if no plane found

        // Store this plane index
        Iterator->Plane = i;
```

We mentioned earlier that it was possible for floating point inaccuracies to cause one more of the vertices in this polygon to be slightly misaligned, causing this polygon to twist to a small degree. Herein lies one of the techniques that we have implemented to help improve the accuracy of the BSP tree compiler.

In the following code we begin to loop through each of the vertices contained within the current polygon. For each vertex encountered here we first calculate the distance at which that vertex lies from the new or existing plane we assigned to the polygon in the previous steps. We then use this information to push the vertex back along the plane normal by that distance such that it will now lie exactly on the surface of that plane. As you might imagine, once every vertex in the polygon has been processed in this way, any twisting that may have caused incorrect classification results during compilation should now have been resolved.

With this step completed we then display the updated progress information to the user – via the logging interface – and move on to the next polygon.

```
        // Retrieve the plane details
        Normal   = GetPlane(i)->Normal;
        Distance = GetPlane(i)->Distance;

        // Ensure that all vertices are on the selected plane
        for ( unsigned long v = 0; v < Iterator->VertexCount; v++ )
        {
            float result = Iterator->Vertices[v].Dot( Normal ) + Distance;
            Iterator->Vertices[v] += (Normal * -result);

        } // Next Vertex

        // Log – BSP : Update Progress

    } // Next Face
```

Once every polygon in the ‘m_pFaceList’ linked list member has been tested and assigned a plane index, this function finishes simply by reporting the successful completion to the user, and returning control back to the calling function – ‘CompileTree’.

```
    // Log – BSP: Operation Successful

    // Success
    return BC_OK;

}
```

CBSPTree::BuildBSPTree

This recursive function is the very heart of the polygon aligned BSP leaf tree compiler. It is functionally very similar in many respects to those we have implemented in the past. From a high level perspective, the following list provides a general summary of the polygon aligned leaf BSP tree construction process:

1) The first step is to select a polygon from those passed into this function that will be used as the candidate plane for a new node. This polygon should be removed from further consideration as a node plane candidate.

2) Loop through each polygon in the list, including the one selected in the previous step.

3) Classify the current polygon against the node plane and add it to the appropriate child list depending on the result of the classification.

4) If the polygon was found to span the node plane, split the polygon against this plane in order to generate two new polygon fragments. Add each of these fragments to the appropriate child list and release the original polygon.

5) Once each polygon has been processed, determine if there are any remaining candidates in front or behind and attach a new leaf or node depending on the outcome of that test.

6) If new nodes were generated, recurse down into those nodes and begin the process again.

As you can see the general construction process remains the same in principle as that employed by the node BSP tree, and even the kD-tree in part. There are, however, a few intricacies and additional steps not included in this list that we will discover as we take a look at how this function is implemented.

```
HRESULT CBSPTree::BuildBSPTree( unsigned long Node, CBSPFace * pFaceList )
{
    // 49 Bytes including Parameter list (based on __thiscall declaration)
    CBSPFace     *TestFace   = NULL, *NextFace  = NULL;
    CBSPFace     *FrontList  = NULL, *BackList  = NULL;
    CBSPFace     *FrontSplit = NULL, *BackSplit = NULL;
    CBSPFace     *Splitter   = NULL;
    HRESULT       ErrCode    = BC_OK;
    CLASSIFYTYPE  Result;
    int           v;

    // Log – BSP: Update Progress
```

This function accepts two parameters. The first is the index to the current node being generated at this point in the recursive process. When calling the function for the first time, this parameter will contain a value of 0 signifying the root node. The second parameter is the first (or head) item of the linked list containing each of the polygons to be processed and classified at this level in the tree.

With this information to hand, the first task that this function must undertake is to select an appropriate node plane candidate – or splitter polygon – from the list passed to this level of the recursive build procedure. This is achieved with a call to the ‘SelectBestSplitter’ method that we first encountered in lab project 16.1. To this function we pass the current list of available polygons, the number of polygons that we would like to sample for selection as well as the split heuristic constant. The ‘SelectBestSplitter’ function will search through the list specified and select an appropriate polygon from those that has not already been used as a splitter in the past. If everything was successful, a reference to this selected

polygon will be returned from the selection routine and stored in the local 'Splitter' variable for use throughout the remainder of this function.

Now that we have an appropriate polygon that can be used as the candidate plane for the node at this level, this polygon's 'UsedAsSplitter' member is set to true. This is done in order to remove it from consideration during further calls to the 'SelectBestSplitter' function. The index stored in the 'Plane' member of this polygon is also then copied into the match member within the current node.

```
    // Select the best splitter from the list of faces passed
    Splitter = SelectBestSplitter( pFaceList, m_OptionSet.SplitterSample,
                                   m_OptionSet.SplitHeuristic);
    if (!Splitter) { ErrCode = BCERR_BSP_INVALIDGEOMETRY; goto BuildError; }

    // Flag as used, and store plane
    Splitter->UsedAsSplitter = true;
    GetNode(Node)->Plane     = Splitter->Plane;
```

Once we have selected the candidate splitter polygon used to generate the node at this stage we then begin the process of iterating through the list of polygons passed in to this level of the recursive procedure in the same way as we have done several times in the past.

```
    // Begin face iteration....
    for ( TestFace = pFaceList; TestFace != NULL;
          TestFace = NextFace, pFaceList = NextFace )
    {
        // Store plane for easy access
        CPlane3 * pPlane = GetPlane( TestFace->Plane );

        // Store next face, as 'TestFace' may be modified / deleted
        NextFace = TestFace->Next;
```

One of the most beneficial aspects of storing indices to the shared plane information within each polygon is that we can determine whether two faces or nodes are coplanar with one another simply by testing for equality between their plane indices. As you might imagine, this could potentially save us from having to perform hundreds or even thousands of plane distance calculations that might be required when testing each vertex via the 'ClassifyPoly' method. Of course, if the plane indices do not match then we must still perform the more traditional polygon classification step, but the savings gained make this step more than worthwhile. From an accuracy standpoint the benefits gained by performing this simple check are also significant because we do not need to use the more inaccurate vertex classification process to determine those polygons that are coplanar with the separating plane.

In the following code you can see this technique employed. If the polygon we are testing has a plane index that is equal to that of the splitter chosen earlier, then we simply store a value of 'CLASSIFY_ONPLANE' in the result variable. If the plane indices do not match however, then we call the plane's 'ClassifyPoly' method to perform the more traditional classification step.

```
        // Classify the polygon
        if ( TestFace->Plane == Splitter->Plane )
        {
```

```
            Result = CLASSIFY_ONPLANE;

        } // End if uses same plane
        else
        {
            Result = GetPlane(Splitter->Plane)->ClassifyPoly(TestFace->Vertices,
                                                             TestFace->VertexCount,
                                                             sizeof(CVertex));

        } // End if differing plane
```

The co-planar case is arguably the most critical of each of the classification cases which is one of the reasons why the aforementioned plane index test is so beneficial. In this on-plane case we must first determine if the polygon faces in the same direction as the current node plane. In our implementation we can use the plane's 'SameFacing' method to test for this.

If the polygon is found to face in the same direction as this coplanar node then we can automatically remove that polygon from being considered as a node plane candidate in the future. If it was not marked as used at this point, then any node generated from it would simply describe the same separating plane as that of the current node. Once this is done, we must then add the current polygon to the linked list representing those polygons or polygon fragments contained in the front halfspace of the node – the linked list headed by the local 'FrontList' variable.

If the polygon does not point in the same direction as the plane then, as we know, this polygon should *not* be removed from consideration in the leaf based BSP tree, and must go on to generate its own node. This polygon must be added to the linked list representing those polygons or polygon fragments contained in the back halfspace of the node – the linked list headed by the local 'BackList' variable.

```
        // Classify the polygon against the selected plane
        switch ( Result )
        {
            case CLASSIFY_ONPLANE:

                // Test the direction of the face against the plane.
                if ( GetPlane(Splitter->Plane)->SameFacing( pPlane->Normal ) )
                {
                    // Mark matching planes as used
                    if (!TestFace->UsedAsSplitter)
                    {
                        TestFace->UsedAsSplitter = true;

                        // Log – BSP : Update Progress Details

                    } // End if !UsedAsSplitter

                    TestFace->Next  = FrontList;
                    FrontList       = TestFace;
                }
                else
                {
                    TestFace->Next  = BackList;
                    BackList        = TestFace;
```

```
                } // End if Plane Facing
                break;
```

The front classification case is relatively straight forward. If we step into this case it means that the polygon is contained completely within the front halfspace of the node. As a result, this polygon is attached directly to the linked list referenced by the 'FrontList' variable.

```
            case CLASSIFY_INFRONT:
                // Pass the face straight down the front list.
                TestFace->Next      = FrontList;
                FrontList           = TestFace;
                break;
```

In a similar fashion to the front case, if the polygon we are testing is found to be completely contained within the back halfspace of the node's plane then it is simply attached to the linked list referenced by the local 'BackList' variable.

```
            case CLASSIFY_BEHIND:
                // Pass the face straight down the back list.
                TestFace->Next      = BackList;
                BackList            = TestFace;
                break;
```

If we step into the spanning case, we have of course found a situation in which the polygon intersects the current node plane, with a portion of that polygon found to lie on either side. This being the case, we must split this polygon into those two fragments separated by the plane in order to pass the relevant portion of the polygon down either side of the node. Before we can do this however we must first allocate the two new polygon objects that will contain these split fragments. To make the handling of errors a little easier in this application, we have implemented a function named 'AllocBSPFace' – covered shortly – that can be used to retrieve a new 'CBSPFace' instance. This is called twice, storing the two resulting polygon pointers into the 'FrontSplit' and 'BackSplit' variables respectively.

```
            case CLASSIFY_SPANNING:
                // Ensure this is not an invalid operation
                // Allocate new front fragment
                if (!(FrontSplit = AllocBSPFace()))
                {
                    ErrCode = BCERR_OUTOFMEMORY;
                    goto BuildError;
                }
                FrontSplit->Next    = FrontList;
                FrontList           = FrontSplit;

                // Allocate new back fragment
                if (!(BackSplit     = AllocBSPFace()))
                {
                    ErrCode = BCERR_OUTOFMEMORY;
                    goto BuildError;
                }
                BackSplit->Next     = BackList;
                BackList            = BackSplit;
```

Notice in the above code that we have immediately added the new polygon objects directly to the front and back lists. We do this in order to ensure that these fragments are automatically released – along with the others in each list – should an error occur in the following split operation.

Having allocated these two new polygon objects, we can now call the spanning polygon's 'Split' method, passing in the separating plane along with the two new polygon objects to be generated. Should it be successful, this will result in the 'FrontSplit' and 'BackSplit' polygons describing the portions of the original polygon that lay on either side of the plane.

```
                // Split the polygon
                if (FAILED( ErrCode = TestFace->Split(*GetPlane(Splitter->Plane),
                                                      FrontSplit,
                                                      BackSplit))) goto BuildError;
```

Recall in the earlier coverage of the 'BuildPlaneArray' method, we described a mechanism in which the vertices of each polygon were pushed back such that they always touched the surface of the plane they reference. Because the polygon splitting operation is prone to floating point accuracy errors – like most other operations – we perform this operation once again here on both the new front and back polygon fragments. While this is not a critical step in the generation of a polygon-aligned BSP tree, this step may help to greatly improve the accuracy of the compilation process.

```
                // Ensure that all vertices are on the original plane
                for ( v = 0; v < (signed)FrontSplit->VertexCount; v++ )
                {
                    float result = FrontSplit->Vertices[v].Dot( pPlane->Normal ) +
                                   pPlane->Distance;

                    FrontSplit->Vertices[v] += (pPlane->Normal * -result);

                } // Next Vertex

                // Ensure that all vertices are on the original plane
                for ( v = 0; v < (signed)BackSplit->VertexCount; v++ )
                {
                    float result = BackSplit->Vertices[v].Dot( pPlane->Normal ) +
                                   pPlane->Distance;

                    BackSplit->Vertices[v] += (pPlane->Normal * -result);

                } // Next Vertex
```

At this stage, we have generated the two new polygon fragments that describe those portions of the original polygon that lay on either side of the plane. We have also added them to the front and back lists ready to be processed at the next level in the recursive procedure. As a result, we have no further need for the original polygon from which these two fragments were generated and therefore we simply release it here. Having removed this original polygon, and inserted two additional polygons into the compile process, the number of active faces will have increased by one. To this end we increment the 'm_lActiveFaces' member here to ensure that our progress is accurately reported before moving on to the next polygon in the linked list passed at this level in the recursive process.

```
                // Log – BSP: Update Progress Details

                // + 2 Fragments - 1 Original
                m_lActiveFaces++;

                // Free up original face
                delete TestFace;

                break;

        } // End Switch

    } // End while loop
```

With every polygon now having been processed, and the resulting front and back lists populated we can now begin to process this information.

During the earlier coverage of the settings contained within the 'BSPOPTIONS' structure we discussed a technique that could be employed to help ensure the integrity of the solid/empty leaf information within the tree. As mentioned, the 'RemoveBackLeaves' flag contained within the 'm_OptionSet' member allows the application to specify whether or not it would like the BSP compiler to enforce the removal of any polygon fragments which end up behind a node where solid space should exist.

If this process was enabled then we first test the contents of the back list to determine whether or not any further nodes will be generated behind the current one. This achieved with a call to the 'CountSplitters' method that is responsible for totalling those polygons in the specified linked list that have not yet been used as node plane candidates. If this function returns a value equal to 0 after having been passed the back list, then we know that we have reached a node behind which solid space exists. Given that this is the case, the following code then proceeds to delete each of the polygons contained within the back list that would have ended up in this solid space.

```
    // Should We Back Leaf Cull ?
    if ( m_OptionSet.RemoveBackLeaves )
    {
        // If No potential splitters remain, free the back list
        if ( BackList && CountSplitters( BackList ) == 0 )
        {
            // Release illegal polygon fragments
            for ( TestFace = BackList; TestFace; TestFace = TestFace->Next )
            {
                delete TestFace;
                m_lActiveFaces--;

            } // End if
            BackList = NULL;

        } // End if No Splitters

    } // End If RemoveBackLeaves
```

At this stage in the process we have access to the final front and back polygon linked lists. This is the ideal time for us to generate the final bounding box extents for the current node based on this information. As a result, we make two calls to the current node's 'CalculateBounds' method. In the first call we pass the front polygon list to the first parameter, and specify that the node's bounding box should be reset by passing a value of 'true' to the second. Once this function returns the node's 'Bounds' members will contain extents values which describe those polygons in the front list. We must take into account *every* polygon that will exist at some point below the node however. As a result we call this method again, passing in the back polygon list and a value of false to the second parameter which instructs the 'CalculateBounds' function that it should not reset the extents. Instead it will grow the current extents to include those polygons passed.

```
    // Calculate the nodes bounding box
    GetNode(Node)->CalculateBounds( FrontList, true );
    GetNode(Node)->CalculateBounds( BackList, false );
```

When processing the front of the node, we know that there can never be a situation in which no polygon data exists within the linked list referenced by the 'FrontList' variable. This is due to the fact that the polygon used as the node plane candidate will always at least be added to the front list. In the following code block therefore we begin by counting the number of splitters that remain in that list.

Recall that a leaf is normally created at the point when there are no further nodes that can be generated from those polygons contained in the appropriate list – e.g. each polygon has its 'UsedAsSplitter' member set to true or the list contains no polygon data at all. Since the latter of these two cases is not possible in front of the node, this means that there must be at least one polygon stored here but it has already been used.

The first case in the conditional if/else statement in the following code block creates a new leaf structure should there have been no available splitter polygons contained in the front list. This leaf is attached to the current node's 'Front' member in the manner outlined in the 'File Based Data Structures' section in which we discussed how leaves and nodes are referenced in this application. After the leaf has been attached to the node in this way, we then make a call to the 'ProcessLeafFaces' method. To this call we pass the newly generated leaf structure and the front polygon list in order to have this function insert those polygons into the leaf structure on our behalf.

```
    // Is the front list empty?
    if ( CountSplitters( FrontList ) == 0 )
    {
        // Add a new leaf and store the resulting faces
        if (!IncreaseLeafCount())
        {
            ErrCode = BCERR_OUTOFMEMORY;
            goto BuildError;
        }

        GetNode(Node)->Front = -((long)GetLeafCount());
        if (FAILED(ErrCode = ProcessLeafFaces( GetLeaf( GetLeafCount() - 1 ),
                                               FrontList ))) goto BuildError;
    }
```

The second case in this conditional if/else statement is executed if there were polygons in the front list that are still available for use as node plane candidates – e.g. at least one of these polygons has a 'UsedAsSplitter' member set to 'false'. In this case there are obviously further nodes that can be generated in front of this node. As a result, we generate a new node structure, store the index to that new node in the 'Front' member of the current node and finally step down the front of this node by recursively calling the 'BuildBSPTree' function here.

```
    else
    {
        // Allocate a new node and step into it
        if (!IncreaseNodeCount())
        {
            ErrCode = BCERR_OUTOFMEMORY;
            goto BuildError;
        }

        GetNode(Node)->Front = GetNodeCount() - 1;
        ErrCode = BuildBSPTree( GetNode(Node)->Front, FrontList );

    } // End If FrontList

    // Front list has been passed off, we no longer own these
    FrontList = NULL;
    if ( FAILED(ErrCode) ) goto BuildError;
```

If you have read the textbook at this point you should be aware that whenever the list of available polygons that fall *behind* a particular node are depleted, then the space behind this node will describe solid space. Recall that a leaf is normally created at the point when there are no further nodes that can be generated from those polygons contained in the appropriate list – e.g. each polygon has its 'UsedAsSplitter' member set to true or the list contains no polygon data at all. In either case, when the back list is empty, we must attach a leaf to the back of the current node that denotes this solid area. Due to the fact that there are no polygons which remain to be collected into the leaf however, it would be wasteful if we allocated and store a new leaf structure here. Instead, we take advantage of the fact that we are using *indices* to reference any leaves attached to either side of the node in order to store a special case 'solid leaf' constant code defined as follows.

```
#define BSP_SOLID_LEAF 0x80000000
```

This value is equal to the largest *negative* value that can be stored in a signed 32 bit variable. Because this constant defines a negative value, the application will still be able to test for the existence of a leaf attached to the back of a node with a simple '<0' comparison test. In order to determine if a leaf is solid however, the 'Back' member of the node must be tested explicitly for equality with this solid leaf define.

In the following code we first check to see if the list of polygons is empty simply by testing the 'BackList' variable to see if it contains a zero or NULL value. If this is found to be the case then we know that solid space must exist behind this node and as a result we set the current node's 'Back' member value to the 'BSP_SOLID_LEAF' constant defined earlier.

```
    // Is the back list empty?
    if ( !BackList )
```

```
    {
        // Set the back as a solid leaf
        GetNode(Node)->Back = BSP_SOLID_LEAF;
    }
```

If the back list *does* contain polygon data at this stage then either the 'RemoveBackLeaves' compile option was disabled by the application, or this backlist simply contains no remaining splitters. In either case, we treat this situation in exactly the same manner as we did with the front list – e.g. create, attach and populate a new leaf structure to the back of this node if there were no splitters remaining or; create and attach a new node behind the current, and recurse into the back of the current node with a further call to the 'BuildBSPTree' function.

```
    else
    {
        // No splitters remaining?
        if ( CountSplitters( BackList ) == 0 )
        {
            // Add a new leaf and store the resulting faces
            if (!IncreaseLeafCount())
            {
                ErrCode = BCERR_OUTOFMEMORY;
                goto BuildError;
            }

            GetNode(Node)->Back = -((long)GetLeafCount());
            if (FAILED(ErrCode = ProcessLeafFaces( GetLeaf( GetLeafCount() - 1 ),
                                                   BackList ))) goto BuildError;

        } // End if no splitters
        else
        {
            // Allocate a new node and step into it
            if (!IncreaseNodeCount())
            {
                ErrCode = BCERR_OUTOFMEMORY;
                goto BuildError;
            }

            GetNode(Node)->Back = GetNodeCount() - 1;
            ErrCode = BuildBSPTree( GetNode(Node)->Back, BackList);

        } // End if remaining splitters

    } // End If BackList

    // Back list has been passed off, we no longer own these
    BackList = NULL;
    if ( FAILED(ErrCode) ) goto BuildError;

    // Success
    return BC_OK;
```

There were several cases within the 'BuildBSPTree' function in which we jumped to the section of code referenced by the label 'BuildError' by using a goto statement. While some might argue that using a

try/catch exception handling method would be more compatible with the C++ standards specification, we have chosen this method within this recursive procedure purely for reasons of efficiency and convenience.

Regardless of how we reach this block of error handling logic, the important point is how each of the front and back lists maintained by this function are being handled in addition to the list of polygons passed into this function. In the following code you can see that each linked list is passed into the 'TrashFaceList' utility function that is designed to add each polygon in the specified list to the 'm_vpGarbage' member array in order for them to be released at a later time. We will discuss why this is the case in the coverage of the 'TrashFaceList' function later in this section.

The final part of this error handling logic returns the error information currently stored in the local 'ErrCode' variable – specified by the main portion of this code – to the calling function. Remember, that because this is a recursive procedure, the function responsible for making the call might actually be this very same one. As a result, we must remember to pay attention to any error codes returned from these recursive calls. If an error was returned from a child recursion, then that call must also begin to clean up and return any applicable error to its parent. This would therefore be repeated until we return all the way back out of each level in the call recursion.

```
BuildError:
    // Add all currently allocated faces to garbage heap
    TrashFaceList( pFaceList );
    TrashFaceList( FrontList );
    TrashFaceList( BackList  );

    // Failed
    return ErrCode;
}
```

CBSPTree::SelectBestSplitter

We encountered the 'SelectBestSplitter' function during our coverage of lab project 16.1. There are however several minor modifications that we must discuss in order to make this function compatible with the BSP leaf tree and our new pre-processing tool.

```
CBSPFace * CBSPTree::SelectBestSplitter( CBSPFace * pFaceList,
                                         unsigned long SplitterSample,
                                         float SplitHeuristic )
{
    unsigned long Score, Splits, BackFaces, FrontFaces;
    unsigned long BestScore = 10000000, SplitterCount = 0;
    CBSPFace *Splitter = pFaceList, *CurrentFace = NULL, *SelectedFace = NULL;
```

Other than using slightly different types (CBSPFace instead of CPolygon for instance) the parameters, local variable list and return value remain similar to that of our previous implementation of this function.

One thing that has changed however is the data that will be passed in to this function. In the previous implementation of the BSP node tree class, the 'SelectBestSplitter' function would be passed a list of only those polygons that had not been linked to nodes in the tree and removed from further processing.

In the BSP leaf tree information, remember that the polygon data is now being passed through the tree in order for it to be collected within the leaves. As a result, this function will now receive even those polygons already used as node plane candidates. For this reason, in the following while loop, this function must ignore any polygon that has its 'UsedAsSplitter' member set to 'true'.

```
    // Traverse the Face Linked List
    while ( Splitter != NULL )
    {

        // If this has NOT been used as a splitter then
        if ( !Splitter->UsedAsSplitter )
        {
            // Create testing splitter plane
            CPlane3 SplittersPlane( Splitter->Normal, Splitter->Vertices[0] );
```

The only remaining difference of any significance within this function is the way in which the splitter plane is generated and tested. In our previous implementation we made use of the D3DXPLANE structure in conjunction with our collision library's 'PolygonClassifyPlane' function. In this lab project however, we make use of the new 'CPlane3' math utility class and its associated 'ClassifyPoly' routine to do the same job.

Other than these minor differences, the mechanism by which a polygon is selected as a potential splitter remains the same. As a result the remainder of this function is included only in the interest of completeness.

```
            // Test against the other poly's and count the score
            CurrentFace = pFaceList;
            Score = Splits = BackFaces = FrontFaces = 0;
            while ( CurrentFace != NULL )
            {
                CLASSIFYTYPE Result =
                         SplittersPlane.ClassifyPoly(CurrentFace->Vertices,
                                                     CurrentFace->VertexCount,
                                                     sizeof(CVertex) );
                switch ( Result )
                {
                    case CLASSIFY_INFRONT:
                        FrontFaces++;
                        break;

                    case CLASSIFY_BEHIND:
                        BackFaces++;
                        break;

                    case CLASSIFY_SPANNING:
                        Splits++;
                        break;

                    default:
                        break;

                } // switch
```

```
                CurrentFace = CurrentFace->Next;

            } // Next Face

            // Tally the score (modify the splits * n )
            Score = (unsigned long)((long)abs( (long)(FrontFaces - BackFaces) ) +
                                                    (Splits * SplitHeuristic));
            // Is this the best score ?
            if ( Score < BestScore)
            {
                BestScore    = Score;
                SelectedFace = Splitter;
            } // End if better score

            SplitterCount++;

        } // End if this splitter has not been used yet

        Splitter = Splitter->Next;

        // Break if we reached our splitter sample limit.
        if (SplitterSample != 0 && SplitterCount >= SplitterSample && SelectedFace)
            break;

    } // Next Splitter

    // This will be NULL if no faces remained to be used
    return SelectedFace;
}
```

CBSPTree::CountSplitters

This method serves as utility function utilized by the 'BuildBSPTree' function to quickly count the number of polygons that have not yet been selected for node plane candidacy within the specified 'CBSPFace'object linked list.

Using the same linked list iteration logic we have employed several times before in this class, the 'CountSplitters' function simply tests the value of each polygon's Boolean 'UsedAsSplitter' member. If this member contains a value of 'false' this indicates that the polygon has not yet been used in the generation of a node. Should this be the case, the counter describing the total number of *unused* polygons is then incremented. This value is maintained within the local 'SplitterCount' variable declared at the top of this function and initialized with a default value of 0. Once each polygon within the linked list has been tested, this function then returns to the calling function passing back this total number of unused polygons.

```
unsigned long CBSPTree::CountSplitters( CBSPFace * pFaceList ) const
{
    unsigned long SplitterCount = 0;
    CBSPFace     *Iterator      = pFaceList;

    // Count the number of splitters
    while ( Iterator != NULL )
```

```
    {
        if ( !Iterator->UsedAsSplitter ) SplitterCount++;
        Iterator = Iterator->Next;

    } // End If

    // Return number of splitters remaining
    return SplitterCount;
}
```

CBSPTree::ProcessLeafFaces

As we already know at this point, the 'ProcessLeafFaces' method is called by the 'BuildBSPTree' function in order to have the polygons contained within the linked list (passed into the second parameter) added to the specified leaf (passed into the first). However, what may not initially have been clear is the reason behind why this was necessary.

As we know, there are two types of results which we can obtain from the BSP tree compiler. Either the leaves reference those polygons that may have been split during the compilation process, or they reference the original unsplit polygons stored in the tree's final polygon array at the start of the process.

This function wraps the logic required for both of these types of tree result. The first of these cases is the non-split resulting tree type. If this type of tree was selected by the application then the polygon linked list passed to this function is sent straight through to the leaf's 'BuildFaceIndices' function. Recall that the polygon list sent to us from the 'BuildBSPTree' function contains those polygons that have been split against and passed through the tree during the compilation process. These polygon fragments each store a value in their 'OriginalIndex' member that references the original unsplit polygon already stored in the tree's polygon array. As a result, the indices for the original unsplit polygons are added directly to the leaf. Because of the fact that we are only interested in the original unsplit versions of those polygons passed to this function in the non-split case, the polygons contained in this list are released with a call to the 'FreeFaceList' utility method.

```
HRESULT CBSPTree::ProcessLeafFaces( CBSPLeaf * pLeaf, CBSPFace * pFaceList )
{
    HRESULT ErrCode = BC_OK;

    // Depending on the tree type we may need to store these resulting faces
    if ( m_OptionSet.TreeType == BSP_TYPE_NONSPLIT )
    {
        // Add these faces to the leaf
        if ( FAILED( ErrCode = pLeaf->BuildFaceIndices( pFaceList )))
             return ErrCode;

        // Release the faces
        FreeFaceList( pFaceList );

    } // End if NSR Type
```

The alternative case is the standard split resulting tree in which the polygon data that has been passed through the tree – and potentially split into many additional polygons – is to be stored in the tree's final

polygon array. As we loop through the list of polygons passed to this function in the split treetype case, we simply allocate a new element in the 'm_vpFaces' array with a call to the 'IncreaseFaceCount' method. Into this element we place the actual pointer to the current polygon object that has been used in the tree compilation process. Finally we update its 'OriginalIndex' member with its own position in the tree's polygon list before passing each of these polygons into the leaf's 'BuildFaceIndices' function as before.

```
    else
    {
        // Split tree, we need to store the face pointers
        CBSPFace * Iterator = pFaceList;
        while ( Iterator != NULL )
        {
            if (!IncreaseFaceCount()) return BCERR_OUTOFMEMORY;
            SetFace( GetFaceCount() - 1, Iterator );

            // Set our original index to the final position it
            // is stored in the array (Split type only)
            Iterator->OriginalIndex = GetFaceCount() - 1;
            Iterator = Iterator->Next;

        } // Next Face

        // Add these faces to the leaf
        if ( FAILED( ErrCode = pLeaf->BuildFaceIndices( pFaceList )))
                    return ErrCode;

    } // End if Split Type

    // Success
    return BC_OK;
}
```

CBSPTree::IncreaseNodeCount / IncreaseLeafCount / IncreasePlaneCount

There are four main utility functions defined within the 'CBSPTree' class that are responsible for resizing the various STL vector members maintained by each instance of this class. The three functions outlined here are each responsible for increasing the size of the node, leaf and plane arrays by one element in each call. The fourth function of this type is responsible for resizing the polygon array but this is implemented in a slightly different manner and is covered independently.

The first of these is the 'IncreaseNodeCount' method. This function is used to allocate and insert a new node object at the end of the BSP tree's internal node array. Due to the fact that this function may be called many times to add a large number of nodes during the BSP compilation process, we use the standard capacity threshold resizing logic in order to prevent this array from being resized / reallocated for every node we add. Put simply, if the total number of elements in the vector *including* the new node is found to exceed its current internal capacity, then we will proceed to allocate additional space by calling the vector's 'Reserve' method. To this method we specify a capacity equal to the current number of items stored in the vector plus an additional number of slack elements into which the vector can grow.

In this way, the vector will only ever be physically resized each time we have added a number of nodes equal to this additional slack space. The number of additional elements reserved in this function is defined by the constant shown below.

```
#define BSP_ARRAY_THRESHOLD 100
```

Once we have reserved any required number of elements in the 'm_vpNodes' member, we can then allocate the new node, and add it to this member with a call to the STL vector's 'push_back' method.

```
bool CBSPTree::IncreaseNodeCount()
{
    CBSPNode *NewNode = NULL;

    try
    {
        // Resize the vector if we need to
        if (m_vpNodes.size() >= (m_vpNodes.capacity() - 1))
        {
            m_vpNodes.reserve( m_vpNodes.size() + BSP_ARRAY_THRESHOLD );
        } // End If

        // Allocate a new Node ready for storage
        // Note : VC++ new does not throw an exception on failure (easily ;)
        if (!(NewNode = new CBSPNode)) throw std::bad_alloc();

        // Push back this new Node
        m_vpNodes.push_back(NewNode);
    } // Try vector ops
```

Each of these functions also implements exception handling logic similar to that shown below. This prevents the BSP compiler from having to wrap each insert operation with its own exception handling logic because any exceptions will have been handled already.

```
    // Catch Failures
    catch (std::bad_alloc)
    {
        return false;
    }
    catch (...)
    {
        if (NewNode) delete NewNode;
        return false;
    } // End Catch

    // Success
    return true;
}
```

We won't go into detail about the remaining two 'Increase*' methods as their functionality is essentially identical. In the interest of completeness however the code to the 'IncreaseLeafCount' method is shown below. This function is used to allocate and insert a new leaf object at the end of the BSP tree's internal leaf array.

```
bool CBSPTree::IncreaseLeafCount()
{
    CBSPLeaf *NewLeaf = NULL;

    try
    {
        // Resize the vector if we need to
        if (m_vpLeaves.size() >= (m_vpLeaves.capacity() - 1))
        {
            m_vpLeaves.reserve( m_vpLeaves.size() + BSP_ARRAY_THRESHOLD );
        } // End If

        // Allocate a new leaf ready for storage
        // Note : VC++ new does not throw an exception on failure (easily ;)
        if (!(NewLeaf = new CBSPLeaf)) throw std::bad_alloc();

        // Push back this new leaf
        m_vpLeaves.push_back(NewLeaf);
    } // Try vector ops

    // Catch Failures
    catch (std::bad_alloc)
    {
        return false;
    }
    catch (...)
    {
        if (NewLeaf) delete NewLeaf;
        return false;
    } // End Catch

    // Success
    return true;
}
```

Finally, the BSP tree class also defines the 'IncreasePlaneCount' method. This function is used to allocate and insert a new plane object at the end of the BSP tree's combined plane array.

```
bool CBSPTree::IncreasePlaneCount()
{
    CPlane3 *NewPlane = NULL;

    try
    {
        // Resize the vector if we need to
        if (m_vpPlanes.size() >= (m_vpPlanes.capacity() - 1))
        {
            m_vpPlanes.reserve( m_vpPlanes.size() + BSP_ARRAY_THRESHOLD );
        } // End If

        // Allocate a new plane ready for storage
        // Note : VC++ new does not throw an exception on failure (easily ;)
        if (!(NewPlane = new CPlane3)) throw std::bad_alloc();
```

```
        // Push back this new plane
        m_vpPlanes.push_back(NewPlane);
    } // Try vector ops

    // Catch Failures
    catch (std::bad_alloc)
    {
        return false;
    }
    catch (...)
    {
        if (NewPlane) delete NewPlane;
        return false;
    } // End Catch

    // Success
    return true;
}
```

CBSPTree::IncreaseFaceCount

As with each of the memory management functions covered so far, this function increases the capacity of the BSP trees internal polygon array using a similar threshold mechanism. There is one major difference with this function when compared to those discussed previously. In this function we do *not* allocate and insert a new 'CBSPFace' object. Instead we simply push a NULL pointer value onto the end of the 'm_vpFaces' STL vector. This behavior was selected due to the fact that in the majority of cases, any polygon data that will be stored within the 'm_vpFaces' array has already been allocated prior to the compilation step. As a result, any existing polygon objects would simply need to be transferred into this array with a call to the 'SetFace' accessor function.

```
bool CBSPTree::IncreaseFaceCount()
{

    try
    {
        // Resize the vector if we need to
        if (m_vpFaces.size() >= (m_vpFaces.capacity() - 1))
        {
            m_vpFaces.reserve( m_vpFaces.size() + BSP_ARRAY_THRESHOLD );

        } // End If

        // Push back a NULL pointer
        m_vpFaces.push_back( NULL );

    } // Try vector ops

    // Catch Failures
    catch (...)
    {
        return false;

    } // End Catch
```

```
    // Success
    return true;
}
```

CBSPTree::AllocBSPFace

The ‘AllocBSPFace’ method is yet another utility function designed to make the job of implementing the more comprehensive BSP leaf tree compilation process a little less clumsy. This function is designed simply to allocate a single ‘CBSPFace’ object that will be passed back via the return value. This could of course be achieved simply by allocating the polygon object within the calling function using the ‘new’ operator. However, rather than have the compiler wrap each allocation attempt in exception handling logic, this function does that on its behalf.

The single parameter accepted by this function is used to specify any ‘CFace' object that should be duplicated into the new ‘CBSPFace’ instance using the overloaded constructor of that class.

```
CBSPFace * CBSPTree::AllocBSPFace( const CFace * pDuplicate )
{
    CBSPFace * NewFace = NULL;

    try
    {
        // Call the corresponding constructor
        if (pDuplicate != NULL)
        {
            NewFace = new CBSPFace( pDuplicate );
        }
        else
        {
            NewFace = new CBSPFace;
        } // End If pDuplicate

        // Note : VC++ new may not throw an exception on failure (easily ;)
        if (!NewFace) throw std::bad_alloc();

    } // End Try

    catch (...) { return NULL; }

    // Success!
    return NewFace;
}
```

CBSPTree::FreeFaceList

The ‘FreeFaceList’ method is a utility function designed to **immediately** release any polygon data stored within the linked list of ‘CBSPFace’ object pointers passed to its single parameter. We observed this method being used during the earlier coverage of the ‘ProcessLeafFaces’ function.

In the following code we can see how this function simply iterates through each element in the specified linked list using the traversal logic we have seen throughout this application. For each polygon encountered in this loop we must first make a duplicate of the pointer stored within that polygon's 'Next' member used to iterate to the next polygon in the list. We must take this action first due to the fact that the current iterator polygon will be deleted in the very next step, rendering that polygon and its members invalid. Before moving on to the next polygon using this duplicated 'NextFace' pointer, we decrement the BSP tree's 'm_lActiveFaces' member due to there now being one less active polygon within the BSP compilation process.

Once the loop completes, this function simply returns control back to the calling function having released every polygon in the list passed.

```
void CBSPTree::FreeFaceList( CBSPFace * pFaceList )
{
    // Free up the linked list
    CBSPFace * TestFace = pFaceList, *NextFace = NULL;
    while ( TestFace != NULL )
    {
        NextFace = TestFace->Next;
        delete TestFace;
        m_lActiveFaces--;
        TestFace = NextFace;
    } // Next Face
}
```

CBSPTree::TrashFaceList

Whenever an error occurs during the construction of the tree, there are most likely many references to the original polygons and new polygons alike that exist in one or more of the lists contained on the stack at each level in the recursion. Under failure conditions, it will be difficult to safely release any allocated polygon data as the recursion unwinds due to the fact that we may attempt to release the same polygon instance on more than one occasion. This would of course cause an illegal memory access and a whole new problem to deal with.

To ensure that we can safely release each of the polygon objects referenced or created by the compile process up to the point of failure, any remaining 'CBSPFace' object pointers can be inserted into the garbage collection member array by passing them to this function. As the code iterates through each of the polygons passed into this procedure, we first check to see if a pointer to that particular polygon already exists within the 'm_vpGarbage' STL vector. Only if no duplicate pointer is found to exist will this function add that polygon to the list. Once the recursive compile process has returned all the way back out to the original calling function, each of the polygons added to the garbage collection list by this function can then safely be released.

```
void CBSPTree::TrashFaceList( CBSPFace * pFaceList )
{
    CBSPFace * Iterator = pFaceList;

    while ( Iterator != NULL )
    {
```

```
        // If it doesn't already exist then add it to the list
        if (std::find(m_vpGarbage.begin(), m_vpGarbage.end(),
            Iterator) == m_vpGarbage.end())
        {

            try
            {
                // Resize the vector if we need to
                if (m_vpGarbage.size() >= (m_vpGarbage.capacity() - 1))
                {
                    m_vpGarbage.reserve( m_vpGarbage.size() +
                                         BSP_ARRAY_THRESHOLD );
                } // End If

                // Push back this new item
                m_vpGarbage.push_back(Iterator);

            } // End Try

            // On exception, we can do nothing but bail and leak
            catch (std::exception&) { return; }

        } // End If Exists

        Iterator = Iterator->Next;

    } // Next Face
}
```

The CProcessHSR Module Class

With the BSP compilation topics out of the way, we can now move onto our coverage of the hidden surface removal processing class that is actually executed in advance of the BSP compile process. It was important that we first discussed the core functionality within the BSP compiler class before moving on to this topic because the HSR process relies so heavily on the information generated by the BSP tree compilation process.

```
class CProcessHSR
{
public:
    // Constructors & Destructors for This Class.
             CProcessHSR();
    virtual ~CProcessHSR();

    // Public Member Functions Omitted

private:
    // Private Variables for This Class.
    vectorMesh      m_vpMeshList;       // List of meshes for HSR Processing.
    CMesh          *m_pResultMesh;      // The resulting unioned mesh

    HSROPTIONS      m_OptionSet;        // HSR Options Set
    ILogger        *m_pLogger;          // Logging output interface
    CCompiler      *m_pParent;          // Parent Compiler Pointer
```

```
};
```

Looking at the declaration for the HSR module class, we can see that this one-off processing module is relatively simple when compared to the BSP compiler class / module covered earlier. This class declaration is a more typical example of the kind of one-time processing module classes that we might develop within this application. There are only a few member variables declared here, so let us cover them very briefly.

vectorMesh m_vpMeshList
This member maintains a list of references to each mesh to be merged together within the hidden surface removal process. It should be populated by the application using the 'AddMesh' method defined by this class. This member variable is of the type 'vectorMesh' which is a typedef of a standard STL vector template designed to store a series of 'CMesh' object pointers.

```
typedef std::vector<CMesh*>    vectorMesh
```

CMesh * m_pResultMesh
Once the hidden surface removal process has run to its conclusion, this member variable will store a pointer to the new mesh that resulted from the union operation. This mesh object can subsequently be retrieved by the compiler class with a call to the 'GetResultMesh' method covered shortly.

HSROPTIONS m_OptionSet
This member stores the various settings, specified by the application to inform the compiler how the hidden surface removal operation should be executed. Although this structure contains no settings outside of the 'Enabled' Boolean, this options structure should be set using the 'SetOptions' accessor function in case of future enhancement.

ILogger * m_pLogger
In order for the HSR processor to report progress, error and useful status information to the user, this member variable stores a pointer to the application defined logging class. While the logging feature remains optional, this member can be set with a call to the 'SetLogger' accessor function defined by this class. In this lab project, the logging class instance is passed to this object by the 'CCompiler::PerformHSR' function.

CCompiler * m_pParent
Each compilation module class maintains a member that stores a reference to the parent compiler class that created it. This member is set with a call to the 'SetParent' accessor function defined by each of these classes. This information is used by each of these processing modules in order to test the current state of the compiler to determine if the compile operation has been either paused or cancelled. Although we have omitted much of the logging and progress functionality from the code listings in this workbook, you should take a look at the source code for lab project 16.2 to see this process in action.

HSR Module Accessor Functions

There are only a few accessor functions defined within the HSR processing module class. In each case, the body for these functions can be found in the class declaration contained in the header file, rather than in the source module / .CPP file. We will not go into detail about how each of these functions is implemented, for this you should take a look at the 'CProcessHSR.h' header file contained in the 'Compiler Source' project directory.

The only accessor functions defined by the 'CProcessHSR' class are those that set the various pieces of information required by each compilation process module. These include the process options, a pointer to the parent compiler and a pointer to the class defined logging class. These functions simply store those specified values into the applicable member variables declared by this class.

```
void            SetOptions    ( HSROPTIONS Options    )
void            SetParent     ( CCompiler * pCompiler )
void            SetLogger     ( ILogger * pLogger     )
```

Now that we are familiar with each of this class's member variables, let us move on to examine the main process functionality.

CProcessHSR::AddMesh

If we refer back to the discussion of the 'CCompiler::ProcessHSR' function, we saw how the 'AddMesh' function was called in order to add each of the scene's non-detail meshes to this hidden surface removal module. This method accepts a single parameter into which is passed a single pointer to a 'CMesh' object. This mesh object pointer will simply be added to the end of the 'm_vpMeshList' member vector using the same capacity threshold mechanism that is used within several other functions in this application. Any meshes that are to be considered by this processing module must be specified in advance, prior to calling this object's 'Process' function.

```
bool CProcessHSR::AddMesh( CMesh * pMesh )
{
    try
    {
        // Resize the vector if we need to
        if (m_vpMeshList.size() >= (m_vpMeshList.capacity() - 1))
        {
            m_vpMeshList.reserve( m_vpMeshList.size() + HSR_ARRAY_THRESHOLD );

        } // End If

        // Finally add this mesh pointer to the list
        m_vpMeshList.push_back( pMesh );

    } // End Try Block

    catch (...)
    {
        // Clean up and bail
        m_vpMeshList.clear();
        return false;

    } // End Catch
```

```
    // Success
    return true;
}
```

CProcessHSR::Process

After having added each of the non-detail scene meshes to this module's source mesh array, this function is called in order to begin the actual hidden surface removal process. Before we move onto examine the implementation for this class, let us once again examine the procedure from a high level overview.

The steps involved in performing the hidden surface removal process are as follows:

1) Allocate and compile a BSP tree object for each individual mesh being processed by this module.

2) Step through each mesh (A) within an outer loop.

3) Step through each mesh (B) within an inner loop.

4) Inside the inner loop, skip any meshes that have already been processed. If the current mesh 'B' has not yet been processed, then test for intersection between mesh 'A' and mesh 'B' using a simple broad phase AABB test. If no intersection occurred, these meshes do not need to be unioned.

5) If the bounding boxes of both meshes intersected, perform an accurate intersection test between these two meshes using the BSP tree information.

6) If mesh 'A' and mesh 'B' were found to intersect with one another then first clip mesh 'A' against mesh 'B's tree, removing any parts of that mesh that exist in solid space. Then do the same in reverse such that mesh 'B' is clipped against mesh 'A's tree.

7) Repair any unnecessary splits that occurred during the clipping process for each mesh, and then exit the inner loop.

8) As the last step of the outer loop, add the surviving polygons from mesh 'A' to the resulting mesh being constructed in this process.

9) Move on to the next mesh in the outer loop and return to step 3 until all meshes have been processed in the outer loop.

Now that we are armed with a general idea of how this process should function, let us take a look at how this has process has actually been implemented in this lab project.

```
HRESULT CProcessHSR::Process()
{
    HRESULT     hRet;
```

```
    BSPOPTIONS  Options;
    int         i, a, b;
    CBSPTree ** BSPTrees  = NULL;
    long        MeshCount = m_vpMeshList.size();
    CBounds3    BoundsA, BoundsB;
```

This function accepts no parameters and simply returns an HRESULT value used to report the success or failure of this process to the calling function.

At the start of this function we start by setting up a temporary 'BSPOPTIONS' structure. These will be the settings that we use to compile each of the mini-BSP trees built for every mesh undergoing the union operation. Within this structure we specify that we would like each tree to use the non-split compilation process. This will ensure that each tree stores an identical set of unsplit polygons as those contained in the original mesh. We also enable the 'RemoveBackLeaves' option. Recall that this setting causes the BSP compiler to forcibly remove any polygons that fall behind a terminal node. This effectively guarantees that a solid space indicator is inserted in place of each empty back leaf that might previously have been created. Should a mesh be slightly malformed, this will help to correct any problems that might arise due to the existence of illegal geometry. The remaining options are set to defaults that will provide a reasonable structure and a fast compile time.

```
    // Build the option set which will be used for all Mini-BSP Trees
    ZeroMemory( &Options, sizeof(BSPOPTIONS));
    Options.TreeType          = BSP_TYPE_NONSPLIT;
    Options.Enabled           = true;
    Options.RemoveBackLeaves  = true;
    Options.SplitHeuristic    = 3.0f;
    Options.SplitterSample    = 60;

    // Log – HSR: Building BSP Trees
```

With these options values initialized, we must now allocate the array in which each mesh' BSP tree will be contained. This array of BSP tree pointers is stored in the local 'BSPTrees' variable declared at the top of the function. The mesh into which all the resulting polygon data can be placed must also be created at this point. Recall that this mesh object is stored within the 'm_pResultMesh' member variable.

```
    try
    {
        // Allocate memory for an array pf BSP Trees
        BSPTrees = new CBSPTree*[ MeshCount ];
        if (!BSPTrees) throw BCERR_OUTOFMEMORY;

        // Set all pointers to NULL
        ZeroMemory( BSPTrees, MeshCount * sizeof(CBSPTree*));

        // Allocate our result mesh
        m_pResultMesh = new CMesh;
        if (!m_pResultMesh) throw BCERR_OUTOFMEMORY;
```

The next step in this process is to compile a BSP tree for every mesh. The following block of code demonstrates how this is achieved.

For each mesh in the 'm_vpMeshList' array we allocate a new 'CBSPTree' object instance, and store its pointer in the newly allocated 'BSPTrees' container array. If the allocation was successful, the 'BSPOPTIONS' structure we set up at the start of this function is then passed into the new BSP tree object with a call to its 'SetOptions' method. Before we can compile the BSP tree, we must first pass in the polygon data from the current mesh object via the 'CBSPTree::AddFaces' method. With this process completed, we can finally call the BSP tree 'CompileTree' method in order for the mesh data to be compiled.

```
        // Build a BSP Tree for each mesh.
        for ( i = 0; i < MeshCount; i++ )
        {
            // Allocate a new tree
            BSPTrees[i] = new CBSPTree;
            if (!BSPTrees[i]) throw BCERR_OUTOFMEMORY;
            BSPTrees[i]->SetOptions( Options );

            // Add the faces from this mesh ready for compile
            hRet = BSPTrees[i]->AddFaces( m_vpMeshList[i]->Faces,
                                          m_vpMeshList[i]->FaceCount );
            if (FAILED(hRet)) throw hRet;

            // Compile the BSP Tree
            hRet = BSPTrees[i]->CompileTree();
            if (FAILED(hRet)) throw hRet;

            // Log – HSR : Update Progress

        } // Next Mesh
```

Now that the BSP tree data for each mesh has been compiled, the next task is to determine which of the mesh objects should be unioned together. While it is certainly possible for us to simply pass the polygon data of one mesh through the BSP tree of another, this could potentially cause hundreds of polygons to be split unnecessarily. This might be the case even if those two mesh objects were on opposite sides of the level. It is logical to assume therefore that if we can find any two meshes that intersect with one another then these two meshes should be unioned.

We search for these intersecting cases by first creating an outer loop that iterates through each of the meshes currently stored in the processor class. We will call this the source mesh. For each source mesh, we must then proceed to iterate through the mesh list once again in order to search for a another mesh that might intersect with the source. We should obviously ignore any cases in which both meshes are the same. However, it is also important to skip over any mesh that has already been processed – e.g. it was a source mesh at an earlier point and has already clipped and been clipped by every other mesh in the scene. Whenever a mesh has been fully processed within the outer loop, the BSP tree for this mesh will be released and set to NULL. If this is found to be the case in the inner loop, then we can safely ignore that mesh.

```
        // Log – HSR: Clipping Meshes for HSR
```

```
        // Now do the actual Union (HSR) clipping operations
        for ( a = 0; a < MeshCount; a++ )
        {
            // Log – HSR: Update progress

            // Skip any NULL bsp objects
            if (!BSPTrees[a] || BSPTrees[a]->GetFaceCount() == 0) continue;

            // Clip against all other meshes
            for ( b = 0; b < MeshCount; b++ )
            {
                // Skip any NULL bsp objects (i.e. has already been clipped)
                if (!BSPTrees[b] || BSPTrees[b]->GetFaceCount() == 0) continue;

                // Don't Boolean Op the mesh with itself
                if (a == b) continue;
```

At this stage we have a reference to two different meshes, neither of which has ever been clipped against the other. To get a rough idea of whether or not these two meshes intersect, we perform a quick broad-phase rejection step by testing for intersection between the bounding boxes of each mesh's BSP tree. If no intersection between these larger bounding boxes was found to have occurred, then we know that the meshes cannot possibly be intersecting. In this case we can continue on to test the next mesh in the inner loop.

If an intersection was found during this broad-phase test, then it is possible that the two meshes also intersect. To make sure that these two meshes really do intersect, we then perform a more accurate intersection test with a call to the 'CBSPTree::IntersectedByTree' function. We will cover this in detail shortly. For now all we need to know is that this test provides a more accurate result, using the BSP tree information constructed from the mesh data itself. Again, if this function indicated that there was no intersection, we can ignore this mesh combination and continue on to the next iteration of the inner loop.

```
                // Skip if the two don't intersect
                BoundsA = BSPTrees[a]->GetBounds();
                BoundsB = BSPTrees[b]->GetBounds();
                if ( !BoundsA.IntersectedByBounds( BoundsB,
                                                   CVector3( 0.1f, 0.1f, 0.1f )))
                    continue;

                if ( !BSPTrees[a]->IntersectedByTree( BSPTrees[b] ) )
                    continue;
```

If we reached this point in the function then it is safe to assume that these two meshes were likely to be intersecting and should be clipped. This is done in such a way that the portions of each mesh are removed if they are contained within the solid space of the other. We perform this clipping operation using the 'ClipTree' method of each BSP tree object. Notice however that the 'ClipTree' procedure accepts a pointer to another BSP tree as its first parameter. In fact we will not be touching any of the original mesh data during the HSR process because these meshes still belong to the compiler object. Instead we will be manipulating only the polygon data stored within each BSP tree object. Once the tree has been compiled, the data stored in its internal polygon array has no bearing on the actual structure of

the tree itself. As a result we can clip and manipulate this data as many times as we like without compromising the integrity of the tree.

If we look at the two calls made to the 'ClipTree' function, we can see that we first call this function on the tree created from the mesh selected in the outer loop, passing in the tree from the inner loop. In the next call this situation is reversed. In both cases we pass a value of 'true' to the second 'ClipSolid' parameter which forms the basis of the CSG union operation. Finally, pay special attention to the value passed to the final 'RemoveCoPlanar' parameter in each case. This will be important when we come to discuss the ClipTree function shortly.

If you have read through the textbook at this point, you should be aware that the CSG process can sometimes split polygons even if neither of the resulting fragments are ever deleted. For this reason we call each BSP tree objects' 'RepairSplits' method which is designed to undo these unnecessary splits. Again, we will examine this function a little later on.

```
                // Clip tree a to tree b and vice versa
                BSPTrees[a]->ClipTree( BSPTrees[b], true, false );
                BSPTrees[b]->ClipTree( BSPTrees[a], true, true  );

                // Repair Unrequired Splits
                BSPTrees[a]->RepairSplits();
                BSPTrees[b]->RepairSplits();

            } // Next Mesh b
```

With the BSP tree selected in the outer loop having now been clipped by each intersecting mesh in the scene, we can add the surviving polygons contained within this BSP tree object to the resulting mesh. This is achieved by calling the 'BuildFromBSPTree' we covered earlier. By specifying a value of false to the final parameter in this method, this will result in the polygon data being appended to those polygons already stored in the 'm_pResultMesh' object. Once the polygon data has been extracted from the tree, we then release it in order to free up any allocated resources that are no longer required. The applicable element in the 'BSPTrees' array is also finally set to a value of NULL. This is to ensure that we do not try and access the recently released BSP tree object at some point in the future.

```
            // Append all faces to the result brush
            m_pResultMesh->BuildFromBSPTree( BSPTrees[a], false );

            // Tree a has now been clipped by all other meshes
            delete BSPTrees[a];
            BSPTrees[a] = NULL;

        } // Next Mesh a

    } // End Try Block
```

The following catch block is designed to release any memory allocated by this method before returning the applicable error code to the calling function should an exception have been thrown at any point.

```
    catch ( HRESULT &e )
```

```
    {
        if ( BSPTrees )
        {
            for ( i = 0; i < MeshCount; i++ )
                if ( BSPTrees[i] ) delete BSPTrees[i];
            delete []BSPTrees;

        } // End if we allocated

        // Release the result mesh
        if ( m_pResultMesh ) { delete m_pResultMesh; m_pResultMesh = NULL; }

        // Log – HSR : Report Failure

        return e;

    } // End Catch Block
```

The final step in this process is simply to release any temporary objects and resources that remain allocated at the end of the process. Once this is complete, we finally return to the calling function with a code indicating success.

```
    // Release the BSP Trees
    if ( BSPTrees )
    {
        for ( i = 0; i < MeshCount; i++ ) if ( BSPTrees[i] ) delete BSPTrees[i];
        delete []BSPTrees;

    } // End if we allocated

    // Log – HSR : Report Success

    // Success!
    return BC_OK;

}
```

Once this entire process has been completed, the ‘m_pResultMesh’ mesh object will contain a copy of every polygon fragment that existed only within an empty area of space. With those fragments existing in solid space having been clipped away, this resulting mesh should now contain a valid set of scene polygon data with all hidden surfaces / illegal geometry removed.

CProcessHSR::GetResultMesh

As we know, the goal of the HSR processing module is to merge each of the specified meshes into one single result mesh. In the previous coverage of the ‘Process’ function defined by this class, we saw how the mesh referenced by the ‘m_pResultMesh’ member variable was constructed. This function is designed to be called by the compiler class in order to retrieve this mesh pointer ready for BSP compilation.

```
CMesh * CProcessHSR::GetResultMesh() const
```

```
{
    return m_pResultMesh;
}
```

CBSPTree – Additional CSG Functionality

With the general concepts involved in performing the hidden surface removal process behind us, let us take a look at that additional support functionality provided by the 'CBSPTree' class for performing CSG operations.

CBSPTree::ClipTree

The 'ClipTree' method of the BSP tree class provides the hidden surface removal processor with the functionality needed to perform the union operation used to merge each of the scene meshes together. Much like the 'BuildBSPTree' method discussed earlier in this chapter, this function is a recursive procedure that passes polygon data through the tree, classifying and splitting against each node until it reaches a terminal node within the hierarchy.

```
HRESULT CBSPTree::ClipTree( CBSPTree * pTree, bool ClipSolid, bool RemoveCoPlanar,
                            ULONG CurrentNode, CBSPFace * pFaceList )
{
    // 50 Bytes including Parameter list (based on __thiscall declaration)
    CBSPFace     *TestFace   = NULL, *NextFace  = NULL;
    CBSPFace     *FrontList  = NULL, *BackList  = NULL;
    CBSPFace     *FrontSplit = NULL, *BackSplit = NULL;
    unsigned long Plane      = 0;
    HRESULT       ErrCode    = BC_OK;

    // Validate Params
    if (!pTree) return BCERR_INVALIDPARAMS;

    // Did Someone use or pass in a silly tree ?
    if (pTree->GetFaceCount() < 1 || GetFaceCount() < 1 )
        return BCERR_BSP_INVALIDGEOMETRY;
```

Although there are 5 parameters declared by this function, only the first three should be passed to the initial call. The final two parameters are used only within this function to pass data between recursive calls. Let us briefly examine the purpose of each of these parameters and what should be passed to each of them.

*CBSPTree * pTree*

The first of these parameters is a pointer to another BSP tree object. The actual tree structure of the object passed to this parameter will *not* be used in this call. Only the polygon data stored within this tree will be manipulated. Recall that in order to perform any type of CSG operation on mesh data; each source mesh must have a BSP tree. With the tree also storing the mesh polygon information, it makes sense that we simply accept a pointer to the tree itself rather than the original mesh.

bool ClipSolid

This parameter is used by the calling function to control the circumstances under which clipped polygon fragments will be removed. If a value of ‘true’ is specified, then any source polygon fragments that end up in the solid space of this tree will be discarded. Conversely, if a value of ‘false’ is specified then those fragments ending up in *empty* space will be discarded. This parameter is the primary means by which we can adapt this same procedure for use with any of the CSG operations.

bool RemoveCoPlanar

This third parameter is used by the calling function to control the manner in which coplanar polygons are treated. If you have read the textbook at this point you should be familiar with the problems associated with the coplanar case in CSG operations. If you were performing the union of two meshes that had overlapping coplanar polygons, were you to remove both polygons you would be left with a hole. If however you were to keep both polygons you would introduced illegal geometry into the new mesh. By removing only one and keeping the other however, the overlapping fragment of the one mesh covers the gap left by the one removed. By alternating this value between the two calls needed to clip both meshes, this same behavior will be observed.

ULONG CurrentNode

This parameter contains the index of the current node in this BSP tree against which we are classifying the source polygon data. This parameter is optional and defaults to a value of 0 (the root node). It should not be specified in the initial call to this function.

*CBSPFace * pFaceList*

This fifth and final parameter is the head / root element in the linked list containing those polygons being clipped during this procedure. In much the same way as the construction of the BSP tree, the front and back lists collected at each step will be passed down to further recursions using this parameter. These polygons are actually references to those owned by the BSP tree object passed in as the first parameter. As with the ‘CurrentNode’, this parameter is optional and defaults to a value of NULL. It should not be specified in the initial call to this function.

With each of the parameters covered, let us move on to the primary implementation of this function.

The first real task with which this function is charged is to construct a linked list from each of the polygons contained in the BSP tree passed in as the first parameter. This is only performed if this is the first time in to this function (determined by the default NULL pointer contained with the pFaceList parameter) and will not take place in subsequent recursions. This is achieved simply by looping through each polygon in the tree using its ‘GetFaceCount’ and ‘GetFace’ accessor functions and attaching each polygon to the ‘Next’ pointer of the one before. Building a list of polygons from the source tree in this way allows us to work directly with its polygon data while maintaining the ability to reuse much of the functionality already designed to work with linked lists. Notice in the following code block that we also reset the ‘ChildSplit’ elements of each face to a default value of -1 in this same loop.

```
    // Automatically build lists
    if ( pFaceList == NULL )
```

```
    {
        // Build our sequential linked list
        for ( UINT i = 0; i < pTree->GetFaceCount(); i++ )
        {
            pTree->GetFace(i)->ChildSplit[0]= -1;
            pTree->GetFace(i)->ChildSplit[1]= -1;
            if ( i > 0 ) pTree->GetFace(i - 1)->Next = pTree->GetFace(i);

        } // Next Face

        // Reset last face just to be certain
        pTree->GetFace(i - 1)->Next = NULL;
        pFaceList = pTree->GetFace(0);

    } // End If No Faces
```

Now that we have access to either the linked list that has just been built in the first call to this function, or the linked list passed in subsequent recursive calls, we now begin to iterate through each polygon in this list. The first thing we must do in this loop is to make a backup of the pointer stored in that polygons 'Next' member. It is important that we do this here because the current polygon may be deleted or modified during this operation. It is also imperative that we ignore any polygons in the list that have been marked as deleted.

```
    // Select node plane and classify / send the list through the tree
    Plane = GetNode( CurrentNode )->Plane;

    for ( TestFace = pFaceList; TestFace; TestFace = NextFace )
    {
        // Store next face, as 'TestFace' may be modified / deleted
        NextFace = TestFace->Next;

        // Skip this polygon it has been deleted in some previous csg op
        if ( TestFace->Deleted ) continue;
```

The next step in the operation, much as in the BSP compile process, is to classify this polygon against the current node's plane and take any appropriate action based on the result.

The first case we deal with here is those polygons from the source BSP tree that are coplanar with the current node. If this polygon is found to be coplanar with the node and points in the same direction, then there is some additional logic that must be applied here to resolve the overlapping polygon problem we outlined earlier. Thankfully however, the solution is relatively simple. If the value passed to the 'RemoveCoPlanar' parameter is true then we simply attach this polygon to the back list being constructed in this call. If the value is false then we add it to the front list instead. Hopefully you can see why this is the case but in summary; should the polygon used to create the node in this tree overlap with the current polygon from the source tree, this source polygon would eventually end up in solid space and be clipped away. Conversely, passing it down the front would push it into an empty area and it would survive. In either case, should the polygon and the node face in opposite directions, the polygon should be added to the back list regardless.

```
        // Classify the polygon against the selected plane
        switch ( GetPlane(Plane)->ClassifyPoly( TestFace->Vertices,
```

```
                                                TestFace->VertexCount,
                                                sizeof(CVertex) ) )
        {
            case CLASSIFY_ONPLANE:
                // Test the direction of the face against the plane.
                if ( GetPlane(Plane)->SameFacing( TestFace->Normal ) )
                {
                    if ( RemoveCoPlanar )
                    {
                        TestFace->Next = BackList;
                        BackList       = TestFace;
                    }
                    else
                    {
                        TestFace->Next = FrontList;
                        FrontList = TestFace;
                    }
                }
                else
                {
                    TestFace->Next = BackList;
                    BackList       = TestFace;

                } // End if Plane Facing
                break;
```

If the polygon fragment was classified as being contained completely within the front halfspace of the node, we simply add it to the current front list in this case.

```
            case CLASSIFY_INFRONT:
                // Pass the face straight down the front list.
                TestFace->Next     = FrontList;
                FrontList          = TestFace;
                break;
```

In a similar fashion as the in-front case, should the polygon fragment be completely behind the node then it is added straight to the back list.

```
            case CLASSIFY_BEHIND:
                // Pass the face straight down the back list.
                TestFace->Next     = BackList;
                BackList           = TestFace;
                break;
```

The initial stages of the spanning case are almost identical to that of the ‘BuildBSPTree’ function. The only real difference here is that the split polygon fragments that we generate are added to the *source* BSP tree passed in to the first parameter. Notice how we call the ‘IncreaseFaceCount’ and ‘SetFace’ methods of the ‘pTree’ object, rather than those local to this object.

```
            case CLASSIFY_SPANNING:
                // Allocate new front within the passed tree
                if (!(FrontSplit = AllocBSPFace()))
```

```
                {
                    ErrCode = BCERR_OUTOFMEMORY;
                    goto ClipError;
                }
                if (!pTree->IncreaseFaceCount())
                {
                    ErrCode = BCERR_OUTOFMEMORY;
                    goto ClipError;
                }

                pTree->SetFace( pTree->GetFaceCount() - 1, FrontSplit );

                // Allocate new back fragment within the passed tree
                if (!(BackSplit     = AllocBSPFace()))
                {
                    ErrCode = BCERR_OUTOFMEMORY;
                    goto ClipError;
                }
                if (!pTree->IncreaseFaceCount())
                {
                    ErrCode = BCERR_OUTOFMEMORY;
                    goto ClipError;
                }
                pTree->SetFace( pTree->GetFaceCount() - 1, BackSplit );
```

With these two new polygon objects created, we can now proceed to split the polygon against the node plane, generating the two new polygon fragments from those portions of the original source polygon that lay on either side.

In the spanning case of the BSP compiler, we follow the split operation by deleting the original polygon. During the process of clipping polygons for CSG however, there are cases in which the splitting of polygon data can occur even when neither fragment will end up being deleted. As a result, it is possible to repair the unnecessary splits but only if the original polygon remains in-tact and accessible. For this reason we simply set the polygon's 'Deleted' member to true in this function. This will still have the effect of causing this polygon to be ignored by this function due to the initial check above. The last piece of information we need to update in the original polygon are the 'ChildSplit elements. Into these we place the indices to the two new polygons as they exist in the **source** BSP tree object. This information will be used during the 'RepairSplits' method we will discuss shortly. Finally we add each of the two new fragments to the relevant front or back list ready for future processing.

```
                // Split the polygon
                if (FAILED( ErrCode = TestFace->Split(*GetPlane(Plane),
                                                      FrontSplit,
                                                      BackSplit))) goto ClipError;

                // Just flag as deleted and mark for potential repair
                TestFace->Deleted    = true;
                TestFace->ChildSplit[0] = pTree->GetFaceCount() - 2;
                TestFace->ChildSplit[1] = pTree->GetFaceCount() - 1;

                // Add it to the head of our recursion lists
                FrontSplit->Next = FrontList;
                FrontList        = FrontSplit;
```

```
                BackSplit->Next  = BackList;
                BackList         = BackSplit;

                break;

        } // End Switch

    } // Next Face
```

With every polygon in the face list now processed, we should have fully populated the 'FrontList' and 'BackList' with all relevant data. At this point in the function we begin the actual process of removing any applicable polygons should we have arrived at a leaf.

We begin with those tests performed when the 'ClipSolid' parameter was set to a value of 'true'. As we know, the only place that solid space can exist within our polygon-aligned BSP leaf tree is behind a node. As a result we test the value of the 'Back' member of the current node to see if it is equal to the 'BSP_SOLID_LEAF' constant value. If this is the case then we know that the space behind this leaf is solid and therefore the polygons contained within the back list can be deleted. Remember that we must not physically release the polygon data at any point in this clipping process; we simply set each polygon's 'Deleted' member to a value of 'true'. Once we have finished iterating through each of the polygons in the list, we must finally clear the 'BackList' variable by overwriting it with a value of NULL. This should be done in order to prevent this list of now deleted polygons from being passed down the back of the node later in this function.

```
    // Now onto the clipping
    if ( ClipSolid )
    {
        if ( GetNode(CurrentNode)->Back == BSP_SOLID_LEAF )
        {
            // Iterate through and flag all back polys as deleted
            for ( TestFace = BackList; TestFace; TestFace = TestFace->Next )
                TestFace->Deleted = true;

            // Empty Back List
            BackList = NULL;

        } // End if Back == Solid

    }  // End if clipping solid
```

If the value passed to the 'ClipSolid' parameter is equal to 'false', then we have been instructed to remove those polygon fragments that may have ended up in any *empty* leaf. Unlike the solid leaf case, if the 'RemoveBackLeaf' BSP compilation option has been disabled then empty leaves can exist both in front of and behind any node. For this reason we must test both the 'Front' and 'Back' members of the current node in this case. If an empty leaf is found to be attached to the back of this node, then again each of the polygons in the back list should be marked as deleted. Conversely, if an empty leaf is found to be attached to the front of this node, then each of the polygons in the *front* list should be marked as deleted. In either case, to prevent any further recursion with this data, we must also remember to clear the applicable 'FrontList' or 'BackList' variable once the polygons have been processed as before.

```
    else
    {
        if ( GetNode(CurrentNode)->Back < 0 )
        {
            // Iterate through and flag all back polys as deleted
            for ( TestFace = BackList; TestFace; TestFace = TestFace->Next )
                TestFace->Deleted = true;

            // Empty Back List
            BackList = NULL;

        } // End if Back == Empty

        if ( GetNode(CurrentNode)->Front < 0 )
        {
            // Iterate through and flag all front polys as deleted
            for ( TestFace = FrontList; TestFace; TestFace = TestFace->Next )
                TestFace->Deleted = true;

            // Empty Front List
            FrontList = NULL;

        } // End if Front == Empty

    } // End if clipping empty
```

It is not necessarily the case that we would have encountered leaves at this stage in the recursive process. As a result, our front and back lists may still be fully populated. Due to the fact that we are actually passing data through the tree in order to determine the leaves in which they are contained, it is necessary for us to traverse into any front and back child nodes at this point. In each case we pass in the same parameter information as was passed to this call with the exception of the last two parameters. Into these we pass the relevant front & back node indices, along with the matching 'FrontList' or 'BackList' variable. Once each side of the node have been fully traversed, we finally return from this function and pass control back to the caller.

```
    // Pass down the front of the node
    if ( FrontList && GetNode(CurrentNode)->Front >= 0 )
            ErrCode = ClipTree( pTree, ClipSolid, RemoveCoPlanar,
                                GetNode(CurrentNode)->Front, FrontList);

    if ( FAILED( ErrCode ) ) goto ClipError;

    // Pass down the back of the node
    if ( BackList && GetNode(CurrentNode)->Back >= 0 )
            ErrCode = ClipTree( pTree, ClipSolid, RemoveCoPlanar,
                                GetNode(CurrentNode)->Back, BackList);

    if ( FAILED( ErrCode ) ) goto ClipError;

    // Success!
    return BC_OK;
```

Because this function is not responsible for creating or deleting any polygon objects that exist only within the stack based linked lists, we do not need to perform any kind of garbage collection in the error handling case. Instead, whenever this function skips to this piece of code we simply return the current error code maintained within the 'ErrCode' variable.

```
ClipError:
    // Failed
    return ErrCode;
}
```

Once this function has completed its recursive execution, any polygons that were contained within the source BSP tree passed to the first parameter of this function will now contain a list of all the newly clipped fragments, in addition to the original polygons simply marked as 'Deleted'. However, no modification of the object in which the 'ClipTree' method was called has occurred. For this tree to subsequently be clipped, the HSR processing module would now perform the same operation again but with the trees reversed. This tree should now be passed in to the first parameter of the 'ClipTree' method of the other.

CBSPTree::RepairSplits

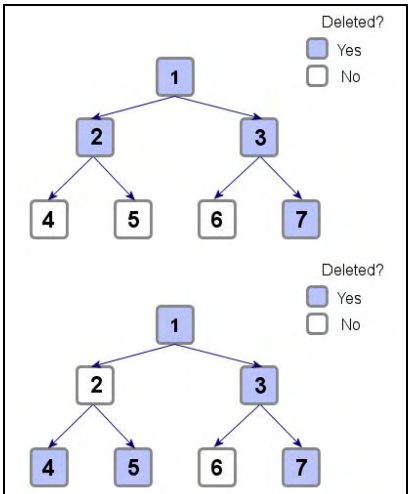


Figure 16.5

The 'RepairSplits' function is an extremely small and yet very fast and elegant way of repairing any unnecessary splits that may have been introduced during the constructive solid geometry clipping process. During the 'ClipTree' process, recall that we store the indices to those polygons created whenever a polygon was split. In addition to this, the original polygon is simply marked as deleted rather than being physically removed.

Referring to the uppermost diagram in figure 16.5, we demonstrate a situation in which the original polygon – shown with an index of 1 – has been split into two child fragments – 2 & 3. This original polygon has then been marked as deleted. Later in the process, each of these child fragments has been split into a further two. Again the source polygons have been flagged as deleted. Finally, polygon 7 gets flagged as deleted during the clipping process. At this point we have a situation in which there are only three polygon fragments that are not marked as deleted. These are the polygons 4, 5 and 6. Notice that polygons 4 and 5 both originated from the same parent – polygon 2. This clearly demonstrates an unnecessary split because both child fragments have survived.

If we now examine the bottom diagram in this same figure we can see what might happen if we repair the unnecessary splits in this scenario. We can see that the child fragments 4 and 5 have now been marked as deleted, and their parent polygon 2 has been resurrected. Its sibling polygon 3 however remains in the deleted state because only one of its child fragments survived.

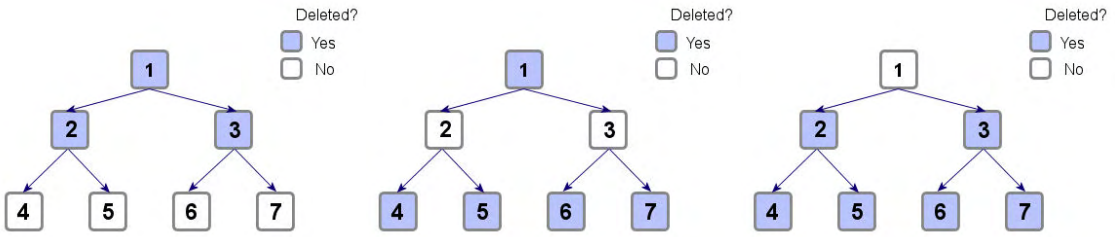


Figure 16.6

Figure 16.6 shows a series of diagrams that demonstrate an alternative scenario. In this case, all 4 of the bottom most children have survived. Both splits generated against polygons 2 and 3 were clearly therefore unnecessary and can be repaired. At this stage, the four child fragments have been marked as deleted and the original fragments 2 & 3 have been restored. Again in this case, these two polygons are fragments resulting from the original polygon 1 being split. Since both child fragments are no longer deleted we can go one step further and restore the original polygon.

As you might imagine, this simple repair process could greatly reduce the number of polygons that might result from an extensive, high-resolution clipping operation. With the theories discussed, let us see how this process is implemented in our 'CBSPTree' class.

The first thing we must do in this function is set up a loop that starts with the last polygon in the BSP tree's polygon array, and works **backwards** to the beginning. If you think back to figure 16.6 remember that we were able to perform multiple levels of repair when starting at the four child fragments and working our way up. In the BSP tree's 'ClipTree' function, whenever new polygon fragments are generated they are added to the *end* of the tree's polygon array with the earliest originals at the start.

Once inside this loop, we first test the contents of one of the 'ChildSplit' elements of the current polygon to see if it makes reference to any split fragment. If either of these elements contains a value other than -1 then this means that this fragment was split at some point. Once we know that a polygon was split into two child fragments, we can then retrieve those fragments and test their 'Deleted' member value. If *neither* fragment is marked as deleted then we will be able to repair this fragment. As a result we set the current polygon's 'Deleted' member to false, and each of the child fragments 'Deleted' members to true.

```
void CBSPTree::RepairSplits()
{
    // For each face
    for ( int i = GetFaceCount() - 1; i >= 0; i--)
    {
        // If one isn't -1, neither is the other
        if ( GetFace(i)->ChildSplit[0] != -1)
        {
            // If the two children are valid this split should be repaired
            if ( !GetFace( GetFace(i)->ChildSplit[0] )->Deleted &&
                 !GetFace( GetFace(i)->ChildSplit[1] )->Deleted )
            {
                // Restore the parent, and delete the children
                GetFace( i )->Deleted = false;
```

```
                GetFace( GetFace(i)->ChildSplit[0] )->Deleted = true;
                GetFace( GetFace(i)->ChildSplit[1] )->Deleted = true;

            } // End if Both Valid

        } // End if Has Children

    } // Next Face
}
```

With this polygon repaired, we continue working backward through the array. Eventually we might come to an earlier polygon that references the polygon we had just repaired. This might be deleted once again, along with a sibling, have *its* parent restored. Once this entire process has completed, we should find that in most cases this repair process is extremely effective because of the fact that it is able to perform partial repairs on original polygons.

CBSPTree::IntersectedByTree / IntersectedByFace

The 'IntersectedByTree' method is the first of two companion functions outlined here. This function is designed to be called by the application in order to test for intersection between two BSP tree objects. The first is the object into which the application is calling, and the second is passed in as the single parameter to this function. This is achieved by passing each polygon contained within the specified BSP tree, through the local tree structure to determine if any of those polygons fall into solid space.

This function is however just a wrapper for the second of these two functions – the 'IntersectedByFace' method. Iterating through each polygon in the BSP tree object passed, this function passes that polygon directly into that method and immediately returning a value of true should it report an intersection.

```
bool CBSPTree::IntersectedByTree( const CBSPTree * pTree ) const
{
    // Loop through each face testing for intersection
    for ( ULONG i = 0; i < pTree->GetFaceCount(); i++ )
    {
        if ( IntersectedByFace( pTree->GetFace(i) ) ) return true;

    } // Next Face

    // No Intersection
    return false;
}
```

The 'IntersectedByFace' method is a recursive procedure and serves as the core of this intersection testing process. This function accepts two parameters. The first is a pointer to the single polygon object to be tested and the second is the index to the current node at this level in the hierarchy. The premise of this function is to pass the specified polygon through the tree, and return an 'intersection' result if that polygon was found to exist in solid space.

> **Note:** This intersection test is not 100% accurate and is designed this way for reasons of efficiency. In order to accurately detect when a polygon falls into solid space it should be split against any node plane it was found to be spanning. Because we do not split the polygon in this way, it is possible for a false

intersection to be returned in certain cases. However, this technique should never return a non-intersection result in cases where an intersection definitely occurred.

```
bool CBSPTree::IntersectedByFace( const CFace * pFace, ULONG Node /* = 0 */ ) const
{
    // Validate Params
    if (!pFace || Node < 0 || Node >= GetNodeCount() ) return false;

    int NodeFront = GetNode( Node )->Front;
    int NodeBack  = GetNode( Node )->Back;
```

The first thing we do at the top of this function is to retrieve the values contained in the front and back members of the current node. This is done for convenience purposes so that we don't have to retrieve them each time we need them.

The next step in the operation is to classify this polygon against the current node's plane and take any appropriate action based on the result.

The first case we deal with here occurs when the polygon we are testing is either co-planar with the node plane, or is found to be spanning. It might seem a little strange to combine both the onplane and spanning case as we have below, but in this function we will not be splitting the test polygon. Instead, all we are interested in is whether there is any part of the polygon in front, behind or on both sides of the node plane. If the polygon is found to be spanning or coplanar then it will be considered to exist both in front and behind. As a result, we check the back of the node to see if it describes solid space. If it does then we immediately return true because we have found a potential intersection. If there is no solid leaf behind, we simply pass this polygon first down the front of the current node and then down the back should a child node exist there in each case. If the recursive call to either 'IntersectedByFace' function returns true, we must also return true such that we will step all the way back out of the recursive procedure.

```
    // Classify this poly against the nodes plane
    switch ( GetPlane(GetNode(Node)->Plane)->ClassifyPoly( pFace->Vertices,
                                                           pFace->VertexCount,
                                                           sizeof(CVertex) ) )
    {
        case CLASSIFY_SPANNING:
        case CLASSIFY_ONPLANE:
            // Solid Leaf
            if (NodeBack == BSP_SOLID_LEAF ) return true;
            // Pass down the front
            if (NodeFront >= 0 )
            {
                if ( IntersectedByFace( pFace, NodeFront ) ) return true;
            }
            // Pass down the back
            if (NodeBack  >= 0 )
            {
                if ( IntersectedByFace( pFace, NodeBack  ) ) return true;
            }
            break;
```

The next case is that in which the polygon is contained completely in the front half space of the current node. Since no solid leaf can exist in front of a node, we simply pass the current polygon down the front of this node should another child node exist there. Again, we return true immediately if this recursive informed us of an intersection.

```
        case CLASSIFY_INFRONT:
            // Pass down the front
            if (NodeFront >= 0 )
            {
                if ( IntersectedByFace( pFace, NodeFront ) ) return true;
            }
            break;
```

Similar to the front case, this case occurs when the polygon is completely contained in the back half space of the node. Solid space can exist *behind* a node however, so we test for this and return true if solid space is found to exist there. If not, we pass the polygon down the back of the current node should another child node exist there.

```
        case CLASSIFY_BEHIND:
            // Solid Leaf
            if (NodeBack == BSP_SOLID_LEAF ) return true;
            // Pass down the back
            if (NodeBack  >= 0 )
            {
                if ( IntersectedByFace( pFace, NodeBack  ) ) return true;
            }
            break;

    } // End Classify Switch
```

If we get here then we did not find any intersections at this level in the tree / recursion. As a result we simply return false from this function in this case.

```
    // No Intersection at this level
    return false;
}
```

This function concludes our coverage of the additional CSG support functionality provided by the ‘CBSPTree’ class. Being the only module that remains to be covered, let us now examine the T-Junction repair process module.

The CProcessTJR Module Class

The T-Junction repair process is the last of the three compiler modules we will implement in this lab project. It is thankfully also the most simple. Because the majority of the code within this process is taken directly from our previous implementation of the T-Junction repair process, we will not dwell on this topic for too long. Before we move on to the modifications that we have made to the procedures, let us take a look at the module class declaration.

```
class CProcessTJR
{
public:
    // Constructors & Destructors for This Class.
             CProcessTJR();
    virtual ~CProcessTJR();

    // Public Member Functions Omitted

private:
    // Private Variables for This Class.
    TJROPTIONS      m_OptionSet;        // The TJR option set
    ILogger        *m_pLogger;          // Log output interface.
    CCompiler      *m_pParent;          // Parent Compiler Pointer

    // Private Member Functions Omitted

};
```

The declaration for the CProcessTJR module class is very similar to that of the hidden surface removal class we looked at earlier. Due to the fact that this module works directly on the existing scene data however, there are even fewer member variables.

Although the member variables declared within this class are only those required by every processing module, let us quickly examine how they apply to this class.

TJROPTIONS m_OptionSet
This member stores the various settings, specified by the application to inform the compiler how the T-Junction repair operation should be executed. Although this structure contains no settings outside of the 'Enabled' Boolean, this options structure should be set using the 'SetOptions' accessor function in case of future enhancement.

ILogger * m_pLogger
In order for the TJR processor to report progress, error and useful status information to the user, this member variable stores a pointer to the application defined logging class. While the logging feature remains optional, this member can be set with a call to the 'SetLogger' accessor function defined by this class. In this lab project, the logging class instance is passed to this object by the 'CCompiler::PerformJTR' function.

CCompiler * m_pParent
Each compilation module class maintains a member that stores a reference to the parent compiler class that created it. This member is set with a call to the 'SetParent' accessor function defined by each of these classes. This information is used by each of these processing modules in order to test the current state of the compiler to determine if the compile operation has been either paused or cancelled. Although we have omitted much of the logging and progress functionality from the code listings in this workbook, you should take a look at the source code for lab project 16.2 to see this process in action.

TJR Module Accessor Functions

There are only a few accessor functions defined within the TJR processing module class. In each case, the body for these functions can be found in the class declaration contained in the header file, rather than in the source module / .CPP file. We will not go into detail about how each of these functions is implemented, for this you should take a look at the 'CProcessTJR.h' header file contained in the 'Compiler Source' project directory.

The only accessor functions defined by the 'CProcessTJR' class are those that set the various pieces of information required by each compilation process module. These include the process options, a pointer to the parent compiler and a pointer to the class defined logging class. These functions simply store those specified values into the applicable member variables declared by this class.

```
void            SetOptions    ( TJROPTIONS Options    )
void            SetParent     ( CCompiler * pCompiler )
void            SetLogger     ( ILogger * pLogger     )
```

Now that we are familiar with each of this class's member variables, let us move on to examine the main process functionality.

CProcessTJR::Process

This function is called by the compiler's 'PerformTJR' function in order to begin the actual T-Junction repair process. This process has no pre-requisites in the form of data being added to the module class in advance. Instead, this method will work directly with that scene data passed into its parameters.

Earlier we discussed the fact that this process is not concerned with any of the renderable polygon information such as textures or materials, or even the surface normal. All we need to have access to are the vertices stored within the scene polygons. As a result, the first parameter declared by this function accepts a list of *pointers* to one or more base 'CPolygon' objects. The second parameter specifies the number of polygons contained in that array.

```
HRESULT CProcessTJR::Process( CPolygon ** ppPolys, ULONG PolyCount )
{
    ULONG      i, k;
    CBounds3  *pBounds = NULL;
    CPolygon  *pCurrentPoly, *pTestPoly;

    // Validate values
    if (!ppPolys || !PolyCount) return BCERR_INVALIDPARAMS;
```

In our previous implementation, we took advantage of the compiled tree in order to rapidly retrieve a list of neighbors for any polygon currently being tested. In this case however we cannot guarantee that a BPS tree has even been compiled. As a result, this function must make its own adjacency arrangements. In order to provide a relatively quick means by which we can determine these neighboring polygons, we first build a list of bounding boxes for every polygon passed into this function using the bounding box's 'CalculateFromPoly' function. We also increase the size of each bounding box by a small amount. This is in order to ensure that the neighbor polygon bounding boxes safely overlap.

```
    try
```

```
    {

        // Allocate space for bounding boxes
        if (!(pBounds = new CBounds3[PolyCount])) return BCERR_OUTOFMEMORY;

        // Log – TJR: Pre-compiling adjacency information

        // Calculate polygon bounds
        for ( i = 0; i < PolyCount; i++ )
        {
            // Log – TJR: Update Progress

            // Build polygon bounds
            pBounds[i].CalculateFromPolygon( ppPolys[ i ]->Vertices,
                                             ppPolys[ i ]->VertexCount,
                                             sizeof(CVertex) );

            // Increase bounds slightly for tolerance
            pBounds[i].Min.x -= 0.1f;
            pBounds[i].Min.y -= 0.1f;
            pBounds[i].Min.z -= 0.1f;
            pBounds[i].Max.x += 0.1f;
            pBounds[i].Max.y += 0.1f;
            pBounds[i].Max.z += 0.1f;

        } // Next Bounds
```

With this bounding box information constructed, we can begin the process of searching for two polygons whose bounding boxes intersect. In a similar manner to the hidden surface removal process, we construct two loops here. The first iterates through each polygon in the outer loop, and another iterates through each polygon in the inner loop. Assuming these aren't both the same polygon, we test for intersection between both polygon bounding boxes with a call to the 'CBounds3::IntersectByBounds' method.

```
        // Log – TJR: Repairing T-Junctions

        // Loop through Faces
        for ( i = 0; i < PolyCount; i++ )
        {
            // Log – TJR: Update Progress

            // Get the current poly to test and its vertex array
            pCurrentPoly = ppPolys[ i ];
            if (!pCurrentPoly) continue;

            // Test against every other face in the tree
            for ( k = 0; k < PolyCount; k++ )
            {
                // Don't against test self
                if (i == k) continue;

                // Get the test face and its vertices
                pTestPoly = ppPolys[ k ];
                if (!pTestPoly) continue;
```

```
                // If the two do not intersect then there is no need for testing.
                if ( !pBounds[i].IntersectedByBounds( pBounds[k] ) ) continue;
```

If the two bounding boxes did intersect at this point then we attempt to run the repair process between each of these polygons, first in one direction and then in the other. We then move on to the next polygon in the inner loop.

```
                // Repair against the testing poly
                if (!(RepairTJunctions( pCurrentPoly, pTestPoly )))
                    throw BCERR_OUTOFMEMORY;

                // Now we do exactly the same but in reverse order
                if (!(RepairTJunctions( pTestPoly, pCurrentPoly )))
                    throw BCERR_OUTOFMEMORY;

            } // Next Test Face
```

At this stage, the current polygon in the outer loop has been processed and repaired against every other polygon in the scene and it need not be considered again in the future. As a result, we overwrite its element in the array with a NULL pointer causing the each loop to bypass this element.

```
            // Completely processed
             ppPolys[i] = NULL;

        } // Next Current Face

    } // End Try Block
```

The final part of this function simply implements the exception handling and clean up code common to most of the functions in this application. Assuming no exception is thrown, the T-Junction repair process has now completed successfully and can return a success code to the calling function.

```
    catch ( HRESULT &e )
    {
        // Clean up and return (failure)
        if (pBounds) delete []pBounds;

        // Log – TJR: Report Failure

        return e;

    } // End Catch Block

    // Release used memory
    if (pBounds) delete []pBounds;

    // Log – TJR: Report Success

    // Success!
    return BC_OK;

}
```

Although this wrapper function contains a reasonable amount of logic on its own, the actual process of repairing the T-Junctions is implemented in the 'RepairTJunctions' method covered in earlier lessons.

CProcessTJR::RepairTJunctions

The 'RepairTJunctions' method outlined below is identical to that we have previously covered and implemented. This is with the exception of it having been adapted for use with the new math support classes. As a result of these minor alterations, we have included this code again here for your convenience.

```
bool CProcessTJR::RepairTJunctions( CPolygon *pPoly1, CPolygon *pPoly2 ) const
{
    CVector3   Delta;
    float      Percent;
    ULONG      v1, v2, v1a;
    CVertex    Vert1, Vert2, Vert1a;


    // Validate Parameters
    if (!pPoly1 || !pPoly2) return false;

    // For each edge of this face
    for ( v1 = 0; v1 < pPoly1->VertexCount; v1++ )
    {
        // Retrieve the next edge vertex (wraps to 0)
        v2 = ((v1 + 1) % pPoly1->VertexCount);

        // Store verts (Required because indices may change)
        Vert1 = pPoly1->Vertices[v1];
        Vert2 = pPoly1->Vertices[v2];

        // Now loop through each vertex in the test face
        for ( v1a = 0; v1a < pPoly2->VertexCount; v1a++ )
        {
            // Store test point for easy access
            Vert1a = pPoly2->Vertices[v1a];

            // Test if this vertex is close to the test edge
            // (Also returns out of range value if the point is past the line ends)
            if ( Vert1a.DistanceToLine( Vert1, Vert2 ) < EPSILON )
            {
                // Insert a new vertex within this edge
                long NewVert = pPoly1->InsertVertex( v2 );
                if (NewVert < 0) return false;

                // Set the vertex pos
                CVertex * pNewVert = &pPoly1->Vertices[ NewVert ];
                pNewVert->x = Vert1a.x;
                pNewVert->y = Vert1a.y;
                pNewVert->z = Vert1a.z;

                // Calculate the percentage for interpolation calcs
                Percent = (*pNewVert - Vert1).Length() / (Vert2 - Vert1).Length();
```

```
                // Interpolate texture coordinates
                Delta.x      = Vert2.tu - Vert1.tu;
                Delta.y      = Vert2.tv - Vert1.tv;
                pNewVert->tu = Vert1.tu + ( Delta.x * Percent );
                pNewVert->tv = Vert1.tv + ( Delta.y * Percent );

                // Interpolate normal
                Delta            = Vert2.Normal - Vert1.Normal;
                pNewVert->Normal = Vert1.Normal + (Delta * Percent);
                pNewVert->Normal.Normalize();

                // Update the edge for which we are testing
                Vert2 = *pNewVert;

            } // End if on edge

        } // Next Vertex v1a

    } // Next Vertex  v1

    // Success!
    return true;
}
```

Lab Project Conclusion

This has been a very interesting but difficult lab project to cover. With the introduction of a completely new framework came new and interesting challenges to overcome. With that said, we now have at our disposal a strong foundation application that can be greatly enhanced as you progress further in your studies with us. With the introduction of the polygon-aligned leaf BSP tree and hidden surface removal processes too, we now have some very exciting opportunities opening up to us. In the next lesson we will be taking these concepts even further with the introduction of portals for leaf connectivity information, and PVS for occlusion culling and visibility which is certainly a topic to look forward to.

Lab Project 16.3: BSP Leaf Tree Loading & Rendering

In lab project 16.3, we begin to put together the rendering application that will make use of the information that has been compiled and exported by the new pre-processing tool developed in the previous lab project. In this application we will move back to our integrated spatial partitioning framework which we have been developing in previous lessons and add a new type of tree responsible for handling the solid BSP leaf tree information. Due to the fact that the compilation of the spatial hierarchy is now delegated to an external application, this new tree class will not perform any physical compile process. Instead it will be responsible only for the loading of the tree data from the compiled IWF file and reconstructing that information into a format compatible with our current spatial hierarchy rendering and utility classes.

The Loading Process

Because we have already developed a full scale scene import process within the 'CScene' class, it would be ideal if we could find a way to have this class remain responsible for loading and processing all level geometry and scene entities. If we can achieve this then no large scale modifications will have to be made within the application itself or any part of our existing spatial partitioning framework. In this lab project we will develop a process in which the scene object first processes the selected IWF file in the same manner as it has always done. Any polygon data imported from that file will simply be added to this new tree *loading* object via the 'ISpatialTree' interface as was the case with the tree *compiler* classes. With this polygon data already in place, the new loading class which will be developed in this lesson will simply be responsible for processing the IWF file for a second time. During this stage, the loading code will retrieve and reconstruct **only** the custom BSP tree information contained therein. As far as the application will be concerned, it would appear as if this new tree class has performed a full compilation of the scene geometry in a similar fashion to the quad-tree class for instance. However, all it has really done is to load the hierarchy data from an external file.

Enhanced Visibility State Recording

In addition to developing this new tree loading concept, we will also enhance the mechanism by which we record the visible state of each of the leaves contained in the tree. In previous applications, during the 'ProcessVisibility' update traversal, we visited every leaf in the tree to update its visibility state with a call to its 'SetVisible' method even if that leaf was not visible. In this tree class, we will be adopting a frame counter system similar to that used by our collision detection and response classes, such that the leaf visibility information is invalidated automatically simply by incrementing a centralized counter variable. Whenever a leaf is found to be visible, a new leaf variable will be updated with the current counter value that can simply be tested during rendering to see if it was marked as visible at any point in this frame of the rendering loop. If the 'SetVisible' call was not made during this frame, then the leaf's internal copy of the counter will be out of date and as such we can take this as a sign that the leaf is not visible. This new mechanism unfortunately prevents the application from making a call to the default 'IsVisible' method of any leaf due to us no longer always updating the 'm_bVisible' member of the 'CBaseLeaf' class. As a result we will need to derive a new leaf class from this base leaf, so that we can

override this behaviour and once again restore this function to a working state. This situation will be discussed in more detail as we move on to discuss the new 'CBSPTreeLeaf' class.

As a direct result of using this new counter mechanism however, we no longer have to visit every single leaf in the spatial hierarchy just to mark it as invisible. As you might image this should have a dramatic effect on the performance of the BSP tree rendering process specifically because it often has many more leaves than the previous types of spatial hierarchy we have developed.

This should be a very interesting use of our existing spatial hierarchy technology. So, without further ado, let us move on to cover the implementation of this new type of tree loader object and its associated support classes and routines.

The CBSPTreeNode Class

As we know, the BSP leaf tree is a *binary* partitioning construct that is designed to separate space into two parts – one portion on either side of a separating plane. This principal is of course identical to that of the kD-tree concept we are already familiar with.

Recall that the node structures within both spatial partitioning schemes are required to maintain similar information. Such information includes a separating plane, front and back child nodes and an optional child leaf. There are also some additional pieces of data that are common to both systems – such as the node's bounding box extents – that serve as the basis for the various rendering and traversal optimizations we have integrated in the past.

If we take a look at the declaration for the new 'CBSPTreeNode' class we should notice that it bears a striking resemblance to the 'CKDTreeNode' class we have previously developed.

```
class CBSPTreeNode
{
public:
    // Constructors & Destructors for This Class.
     CBSPTreeNode( );
    ~CBSPTreeNode( );

    // Public Variables for This Class
    D3DXPLANE       Plane;              // Splitting plane for this node
    CBSPTreeNode *  Front;              // Node in front of the plane
    CBSPTreeNode *  Back;               // Node behind the plane
    CBSPTreeLeaf *  Leaf;               // Leaf may be stored here
    D3DXVECTOR3     BoundsMin;          // Minimum bounding box extents
    D3DXVECTOR3     BoundsMax;          // Maximum bounding box extents
    signed char     LastFrustumPlane;   // The frame-to-frame coherance index.

    // Public Members Omitted
};
```

Due to the fact that this node structure is almost identical to that used by both the kD-tree and BSP node tree, we will only briefly recap on each of the member variables declared here.

D3DXPLANE Plane
This member describes the separating plane used in the creation of this node. In the case of the polygon-aligned BSP tree, this will be oriented such that it matches the plane of at least one polygon in the scene. In this new rendering application, the plane data is pulled from our combined plane array – stored in the import file – and placed into the D3DXPLANE typed member that we have relied upon throughout. We will examine this procedure shortly.

CBSPTreeNode * Front
Each node in the BSP tree partitions the scene into pieces which represent the space contained in both its front and back halfspaces. If there are further polygons available to be selected in front of the current node, then our compiler will have recursed into the front list and continued to generate new nodes from that data. The first of these nodes would be attached to this member in exactly the same manner with which we are familiar. In contrast to our compiler tool, our rendering application does not combine the concept of attached child leaves and nodes into one member variable and as such only a child node can be attached here. This is similar to the previous run-time tree types we have implemented prior to this, represented with a type that contains a pointer to another 'CBSPTreeNode' object.

CBSPTreeNode * Back
This member is similar to the previous in that it stores a pointer to any child node that might exist in the back halfspace of the current. Again this is identical in principal to the kD and BSP node tree types we have implemented in the past.

CBSPTreeLeaf * Leaf
Should this be a terminal node, this member will store the child leaf that should be attached here. In the earlier lab projects dealing with spatial hierarchies, we found that *every* terminal node would have a leaf attached to it. In this application we are implementing the run-time portion of the **solid** leaf BSP tree. Recall in lab project 16.2, there were cases in which a particular area of space within the scene was found to be solid. In these cases, no leaf was created or inserted into the tree. These leaves were simply replaced with an indicator that allowed us to identify this fact. As a result, when reconstructing the BSP tree in this application, there may be cases where this member is assigned a value of NULL that is used to signify the same thing. More details will be provided on this subject as we move into covering the actual reconstruction of the hierarchy from file.

D3DXVECTOR3 BoundsMin
This member stores the value that describes the minimum extents of the node's bounding box. Recall that this bounding box will be large enough to contain every node, leaf and polygon that is found to exist anywhere beneath the current node.

D3DXVECTOR3 BoundsMax
Like the previous, this member stores part of the data required to describe the node's axis-aligned bounding box. In this case however, this member stores the *maximum* extents of the box that describes each of the elements contained in this node's subtree.

signed char LastFrustumPlane
We have encountered and discussed this member in previous lessons. Recall that it is used during the 'ProcessVisibility' traversal process when testing the node bounding box against the camera's frustum

planes. This member stores the last frustum plane found to intersect this node's parent bounding box. This is an optional value that can be passed into the camera's 'BoundsInFrustum' function to allow it to optimize the order in which it tests the frustum planes during the processing of a hierarchy such as this.

With each of the member variables discussed, it should be clear that this node class does not declare anything that we have not encountered before with the kD-tree node type. Similar to the 'CKDTreeNode' class, this exposes only a single method. Let us take a look at this now.

CBSPTreeNode::SetVisible

The only method defined by this node class is the 'SetVisible' function. As we know, this function is traditionally called during the 'ProcessVisibility' traversal in each of our previous tree types. This implementation of the BSP tree is no different in that regard. It accepts a single Boolean parameter that specifies whether or not this node and its children are visible, this function then passes that information on to any applicable children. At the top of the function we can see that if there is a leaf stored in this node, that leaf's own 'SetVisible' function will be called – passing in the same visibility status – before returning. If there was no leaf stored here, then this recursive procedure calls the 'SetVisible' function of both its front and back child nodes should either be available. This has the effect of traversing through the tree and updating the visibility status of every leaf contained anywhere beneath the initial node on which this function has been called.

```
void CBSPTreeNode::SetVisible( bool bVisible )
{
    // Set leaf property
    if ( Leaf  ) { Leaf->SetVisible( bVisible ); return; }

    // Recurse down front / back if applicable
    if ( Front ) Front->SetVisible( bVisible );
    if ( Back  ) Back->SetVisible( bVisible );
}
```

As we can see, this simple recursive function remains unchanged from each of the four node classes we have created within the previous spatial hierarchy types.

The CBSPTreeLeaf Class

This class provides the data structures required to represent the leaf items contained within the spatial hierarchy. Again the concepts involved in attaching objects of this type to the tree should be very familiar to us already. However, there is a key difference between the implementation of the leaf concept used in previous spatial hierarchies and that of this new BSP loading class. In each of the previous tree types, recall that we used instances of the 'CBaseLeaf' class exclusively within the tree structure. This base class provides much of the functionality required to store the various pieces of information – such as the polygons and objects contained within that leaf – in addition to the functionality required for rendering much of that data.

As we move forward with the BSP leaf tree concepts, the leaf structure will be required to store and process much more information that is specific to this type of tree. As a result, within the implementation of this spatial hierarchy type, we will begin to create a new leaf class that *derives* from

‘CBaseLeaf’. This is done in order to extend that base with the additional members and functionality required in both this lesson and the next.

With this in mind, let us take a look at this new leaf class declaration before moving on to the initial methods that will be exposed.

```
class CBSPTreeLeaf : public CBaseLeaf
{
public:
    // Constructors & Destructors for This Class.
             CBSPTreeLeaf( CBSPTreeLoader * pTree );
    virtual ~CBSPTreeLeaf( );

    // Public Virtual Functions for This Class (from base).
    virtual bool    IsVisible  ( ) const;
    virtual void    SetVisible ( bool bVisible );

    // Public Variables for This Class
    ULONG           m_nVisCounter;
};
```

As you can see, there is only one member variable declared by this class outside of those inherited from the base ‘CBaseLeaf’ class that we are already familiar with. The purpose of this variable is described below.

ULONG m_nVisCounter
This member relates to the new visibility counting mechanism mentioned earlier in this lesson. Should the leaf be visible in any given render call, this variable will be updated such that it contains the same value as that currently stored within the main tree object. If at any point the leaf is found to be outside of the viewing frustum, this member will *not* be updated and will automatically become out of date as the main counter has been incremented. Because the tree’s counter is incremented every frame, if the value of this member differs from that of the tree object at any point, we know that this leaf was found not to be visible within **this** frame of the render loop.

With this new visibility counting scheme we must alter the way that the ‘CBaseLeaf’ class detects and reports this leaf’s visibility status to the user. This is achieved by overloading the virtual ‘IsVisible’ method such that when the application requests the leaf’s visibility information through a pointer to the ‘ILeaf’ superclass, the function defined in this new ‘CBSPTreeLeaf’ will be called. In addition, we must also provide an overload of the ‘SetVisible’ function. This method is *not* declared within the ‘ILeaf’ interface and should never be called by the application. However, we must overload the ‘CBaseLeaf::SetVisible’ function in order to perform the additional processing that is now required.

CBSPTreeLeaf::CBSPTreeLeaf( )

The first function we need to discuss is the single constructor for this class. This constructor accepts a single parameter as input. This parameter is a pointer to the tree object into which this leaf will be attached. Recall from our earlier discussion of the ‘CBaseLeaf’ class, this *parent* information is placed into an internal member variable and is used to allow the leaf object to access information stored within

the tree itself. Both the member variable and the code to store the input parameter are inherited from this class's base. As a result, we must ensure that the base class constructor is also called, passing to it the same information that this constructor received. The easiest way to achieve this is to simply call the relevant 'CBaseLeaf' constructor by using the initialization list portion of the 'CBSPTreeLeaf' constructor shown below.

```
CBSPTreeLeaf::CBSPTreeLeaf( CBSPTreeLoader * pTree ) : CBaseLeaf( pTree )
{
    // Reset / Clear all required values
    m_nVisCounter   = 0;
}
```

With the information inherited from 'CBaseLeaf' fully initialized by the base class constructor, we can then simply continue to initialize those members added by this class. In this lab project, the only variable we need to reset here is the 'm_nVisCount' member. This is set to '0' in order to ensure that every leaf starts its life in an *in*visible state.

CBSPTreeLeaf::SetVisible

Earlier we talked a little bit about the new mechanism by which we are recording information about whether or not a leaf is visible. This scheme uses a counter variable that describes whether or not the leaf was marked as visible within the most recent call to the tree class's 'ProcessVisiblity' method. In this function and the next, we can see just how this is achieved. Recall that this is the first spatial hierarchy type in which we are deriving a new class in order to customize some of the core functionality provided by the base 'CBaseLeaf'. Because we want to add an additional mechanism by which the visibility status of the leaf is recorded we need to overload the base class' 'SetVisible' function.

In the code block that follows we have included additional code that updates the member variable responsible for recording the most recent frame in which this leaf was found to be visible. If this leaf was specified as being visible then the current visibility frame counter value is requested of the parent tree object to which this leaf is attached. The value returned is subsequently stored in the leaf's 'm_nVisCounter' member variable. If a value of 'false' was passed to this method's single parameter, then we simply reset the visibility counter to 0. Although in this application we will never explicitly call the 'SetVisible' method when a leaf is **not** visible – due to our new updated counter mechanism automatically invalidating such leaves – we include the latter case in order to ensure that the system exhibits the correct behaviour should we need to do so in the future.

```
void CBSPTreeLeaf::SetVisible( bool bVisible )
{
    // Update our current vis counter
    if ( bVisible )
    {
        // Store the current visibility counter
        m_nVisCounter = ((CBSPTreeLoader*)m_pTree)->GetVisCounter();

    } // End if visible
    else
    {
        // Reset the visibility counter
```

```
        m_nVisCounter = 0;

    } // End if not visible
```

With the ‘m_nVisCounter’ member variable either having been updated with the current frame counter, or reset to a value of 0 depending on the state of the ‘bVisible’ parameter, we must now pass on this message to the base class implementation of this function. This is done to allow the ‘CBaseLeaf’ version of this method to perform any necessary steps for adding this leaf to the tree’s internal visible leaf array as well as populating the appropriate buffers prior to rendering.

```
    // Call base class implementation
    CBaseLeaf::SetVisible( bVisible );
}
```

Given this relatively simple logic it should be clear that the ‘m_nVisCounter’ member of any given leaf will only be updated to match that of the tree if this method has been invoked in any specific visibility update traversal. Because the *tree’s* internal visibility counter is automatically incremented with each subsequent call to the ‘ProcessVisibility’ function in every frame, this of course means that if the leaf was not found to be visible at any point, the ‘m_nVisCounter’ variable maintained by that leaf will no longer match. This effectively allows us to mark every leaf as invisible at the start of each frame simply by incrementing that single counter value maintained by the main tree object. This negates the need to explicitly call the ‘SetVisible’ method for each leaf, passing a value of ‘false’ during the visibility traversal as we have done in the past. We will take a closer look at the changes we can make to the ‘ProcessVisibility’ method of the main tree class to improve its efficiency a little later in this lesson.

Due to the fact that we still want the application to be able to query the visibility state of each leaf, we must also override the ‘IsVisible’ method to take this new frame counter mechanism into account.

CBSPTreeLeaf::IsVisible

Should the leaf’s counter variable have been updated in a call to its ‘SetVisible’ method then logically the value stored in the leaf’s visibility counter should match that maintained by the associated tree object. Of course, this will only be the case for the duration of time until the next call to ‘ProcessVisibility’ is made. If the leaf is not subsequently marked as visible in that frame as well, then the value stored in the leaf’s ‘m_nVisCounter’ variable will be out of date. This removes the need for us to traverse every part of the tree simply to set a leaf as *not* being visible.

Given these facts, the ‘IsVisible’ method has been overloaded in this class to return a status of ‘true’ **only** if the leaf was set to visible in this frame of the applications main rendering loop – or more accurately the most recent call to the tree’s ‘ProcessVisibility’ method. This is done by comparing the visibility counter value stored here, to the value currently maintained by the parent tree object we set in the class constructor. If these two values match, then we know that the leaf has been updated in the current frame and the comparison operation shown below will result in a value of true being returned. If this is not the case, then the function will return false.

```
bool CBSPTreeLeaf::IsVisible( ) const
{
```

```
    return (m_nVisCounter == ((CBSPTreeLoader*)m_pTree)->GetVisCounter());
}
```

With our newly overridden leaf tree and node classes implemented in their entirety, let us now begin to examine the core derived tree class that we will by using in this lab project.

The CBSPTreeLoader Class

Like each of the spatial tree classes we have developed in the past, this new class is derived from 'CBaseTree' from which we inherit a large portion of the required storage, management and rendering functionality. The only task that each of these spatial hierarchy classes has essentially been responsible for is to populate the leaf, node and polygon arrays maintained by the base class itself. As we know this was previously achieved by taking the polygon data that has been loaded and passed in to the tree class and compiling that information into a given type of spatial hierarchy. In the case of lab project 16.3, the compilation process has already taken place and we are simply reconstructing that tree into a format compatible with our already existing spatial hierarchy utility and rendering classes. As a result, the majority of the code that we will be implementing within this class revolves around the loading and interpreting of the BSP tree information contained within the source IWF file and simply storing it in the appropriate base class container members. With this in mind, let us take a look at the new 'CBSPTreeLoader' class declaration to see how we might integrate such a concept into the existing hierarchy system.

```
class CBSPTreeLoader : public CBaseTree
{
public:
    // Constructors & Destructors for This Class.
    virtual ~CBSPTreeLoader( );
             CBSPTreeLoader( LPDIRECT3DDEVICE9 pDevice, bool bHardwareTnL,
                             LPCTSTR FileName );

protected:
    // Typedefs, Structures and Enumerators.
    struct iwfNode
    {
        long        PlaneIndex;
        D3DXVECTOR3 BoundsMin;
        D3DXVECTOR3 BoundsMax;
        long        FrontIndex;
        long        BackIndex;
    };

    // Protected Variables for This Class
    CBSPTreeNode * m_pRootNode;         // The root node of the tree
    LPTSTR         m_strFileName;       // File to load

    CBaseIWF       m_FileLoader;        // The IWF parsing object
    iwfNode       *m_pFileNodes;        // Node data loaded from file
    D3DXPLANE     *m_pFilePlanes;       // Plane data loaded from file.
    ULONG          m_nFileNodeCount;    // Number of nodes loaded from file
    ULONG          m_nFilePlaneCount;   // Number of planes loaded from file
```

```
    // Protected Functions Omitted

};
```

There are several new members declared here that we have not yet encountered in any of the spatial hierarchy classes that we have previously developed. Each of these member variables are outlined below.

CBSPTreeNode * m_pRootNode
The 'm_pRootNode' member stores a pointer to the root 'CBSPTreeNode' object of the constructed spatial hierarchy. This is the entry point into the tree and is used to start any traversal operation as we know. This member will be assigned during the tree reconstruction step in the 'BuildTree' function discussed shortly.

LPTSTR m_strFileName
This member stores a duplicated copy of the path and file name for the file to be imported. This filename string should reference the IWF file that has been compiled and exported by the pre-processing tool developed in lab project 16.2. This member is populated in this class's constructor and later used by the 'Build' function during the file import step (covered shortly).

CBaseIWF m_FileLoader
Due to the fact that this tree type does not perform a tree compilation process and instead simply loads the data from the IWF file, we need access to the file data. For this we use the 'CBaseIWF' class provided to us by the 'libIWF' import library. The 'm_FileLoader' member stores an object of this type. Put simply, this object will be used to perform all of the actual parsing of the IWF file and its structure.

By using a series of callback functions, the 'CBaseIWF' object will notify us whenever it encounters a chunk of a type within the file that we are interested in. With the file automatically positioned at the beginning of that applicable data, we can simply read the information using the stream functions provided by the 'CBaseIWF' class. For more information on this class and its methods, refer to appendix A in this chapter. We will see how this can be used to help us load the custom tree data as we move on to discuss the overloaded 'Build' function in addition to the registered callbacks that we must supply in order to read that information.

iwfNode * m_pFileNodes
During the import of the tree information contained in the IWF file, there are cases where we must load the data into temporary structures. The file data that describe the nodes within the BSP tree is just such a case. As we know, each node item stored within the file has a number of dependencies. These include the planes that are contained in the file's combined plane array, the leaf objects that may be attached to the node and of course the front and back child nodes. At the point in which we are loading the data for any given node, we may not yet have imported any plane or leaf items that may be attached to that node. More importantly than either of these is the fact that, until the entire node array has been loaded, the build procedure will not have access to the front and back child node information and will be unable to rebuild the physical tree structure. As a result, we must import the node data into a temporary area until such time as every node has been loaded. Once this has been done, we will then have full access to each node that is contained in the tree in order for us to reconstruct it.

Because of the fact that the structure of the node data stored within the file is different to that of the run-time application, and uses an indexing scheme to reference the applicable node, leaf and plane information, we must create a new structure that will hold the file's information temporarily. Let us take a quick look at the iwfNode structure we have defined in order to achieve this.

struct iwfNode

- *long PlaneIndex*

  As with each of the variables declared within this 'iwfNode' structure, the data stored in this member is loaded directly from the IWF file that is being processed. In particular, this member stores an index into the file's combined plane array that describes the plane on which this particular node lies. This information will be used to retrieve the correct plane data during the reconstruction of the tree into a format suitable for this application.

- *D3DXVECTOR3 BoundsMin*

  This member stores the value that describes the minimum extents of the node's bounding box. Recall that this bounding box will be large enough to contain every node, leaf and polygon that is found to exist anywhere beneath it in the tree.

- *D3DXVECTOR3 BoundsMax*

  Like the previous, this member stores part of the data required to describe the node's axis-aligned bounding box. In this case however, this member stores the *maximum* extents of the box that describes each of the elements contained in this node's subtree.

- *long FrontIndex*

  This member represents the index of the item attached to the front side of the current node stored within the appropriate array. Recall from our earlier discussion of lab project 16.2 that this can either be another child node or a leaf. If the sign of the index is positive then this member references a child node. If it is negative, this signifies that a leaf is attached to this side of the node. Our run-time tree hierarchy is constructed in a slightly different manner, providing additional leaf-nodes to which we attach the physical leaf data, simplifying the traversal process. This is yet another reason why ensuring that the node data is fully loaded and stored in a temporary array before reconstructing the tree is useful to us.

- *long BackIndex*

  The 'BackIndex' member is identical in principal to the aforementioned 'FrontIndex'. The only difference here is that this member is used to reference any leaf or node child that may be attached to the *back* of the current node. Also recall that in the solid tree – such as that compiled by the new pre-processing tool – this member may also store a value equal to that defined by the 'BSP_SOLID_LEAF' constant if the space behind this node is to be considered solid. It is

important to bear this in mind because we will need to recognize this situation when reconstructing the tree hierarchy.

With each of the individual elements in the temporary node structure defined, let us now continue on to discuss the remainder of the member variables declared within the 'CBSPTreeLoader' class.

D3DXPLANE * m_pFilePlanes
This member stores the array of plane data that was compiled and stored in the scene IWF file. Due to the fact that the plane information is stored in a separate linear block from the node data, this member also acts as a temporary container much like the 'm_pFileNodes' array. This array will be populated during import and will be used during the reconstruction of the spatial hierarchy as the basis for the plane data stored at each node. Recall that the node structure within the file stores an index to a specific plane within the combined plane array stored in the file. Since this array will represent that same plane data it should be used as the source for the plane referenced by the 'PlaneIndex' value stored in each temporary 'iwfNode' structure.

ULONG m_nFileNodeCount
This member stores the total number of nodes loaded from file and subsequently stored in the array referenced by the 'm_pFileNodes' member variable. This information is not used directly during the reconstruction of the tree, but it can be used for the purposes of validation to ensure that node data was actually loaded from the source file. This information might also be used to check each node's front and back index variables such that we return gracefully if any index exceeds the boundaries of the array, indicating that the node data stored in the file is malformed or corrupted.

ULONG m_nFilePlaneCount
As with the previous variable, this member stores the total number of planes loaded from file and subsequently stored in the 'm_pFilePlanes' array. Again, this information is not used directly during the reconstruction of the tree but it might be used to provide an extra layer of validation to protect against file corruptions or invalid information.

As we can see, with the exception of the 'm_pRootNode' variable, each of the members contained within this *loader* class exist primarily for the purpose of handling the import and processing of the scene hierarchy data. This information will then be used to reconstruct the hierarchy in a manner compatible with the 'ISpatialTree' interface concept we have been developing. Now that we have a rough understanding of each of the new class member variables, let us now move on to discuss the various methods of this class that are responsible for performing these operations.

CBSPTreeLoader::CBSPTreeLoader( )

We are already familiar with the steps involved in setting up a constructor for a new tree type that has been derived from 'CBaseTree'. As in each of our earlier tree classes, the constructor must accept both a valid Direct3D device, in addition to a boolean flag indicating whether hardware transform and lighting can and should be used when rendering any scene geometry. These two pieces of information must be passed to the base class constructor to allow any vertex and index buffer resources to be created using the correct parameters. In this new 'CBSPTreeLoader' class, this is achieved in a similar manner to those we have previously implemented, by passing these two parameters to the base constructor using

the initialization list portion of the function. The only additional piece of information that is required specifically by our new BSP tree loading class is the name of the file from which the tree data should be imported. This is passed by the application as the third argument to this constructor.

The body of this constructor is very simple and is responsible for initializing the class member variables with their default values. In this particular constructor we must also make a copy of the filename specified by the application. We do this using the '_tcsdup' runtime function in the same way as we have done many times before. It is important that we make a copy of this string here rather than simply storing the specified string pointer in order to ensure that the memory referenced by the 'FileName' parameter is not altered or released before we get a chance to begin importing and building the tree information.

```
CBSPTreeLoader::CBSPTreeLoader( LPDIRECT3DDEVICE9 pDevice, bool bHardwareTnL,
                                LPCTSTR FileName ) :
                                CBaseTree( pDevice, bHardwareTnL )
{
    // Clear required variables
    m_pRootNode       = NULL;
    m_pFileNodes      = NULL;
    m_pFilePlanes     = NULL;
    m_nFileNodeCount  = 0;
    m_nFilePlaneCount = 0;
    m_nVisCounter     = 0;

    // Make a copy of the file name
    m_strFileName = _tcsdup( FileName );
}
```

Once an object of this type has been instantiated, and the relevant pieces of information initialized and stored, the application can then instruct that object to begin building the spatial hierarchy information.

CBSPTreeLoader::GetVisCounter

The 'GetVisCounter' function is a publically accessible class method that allows other methods and objects to gain access to the tree's current visibility 'call' counter. This method is used by the 'SetVisible' and 'IsVisible' methods of the new 'CBSPTreeLeaf' class in order to retrieve the current counter value that has been incremented by a call to 'ProcessVisibility'. This function simply returns the value currently stored in the 'm_nVisCounter' member variable and takes no further action.

```
ULONG CBSPTreeLoader::GetVisCounter( ) const
{
    return m_nVisCounter;
}
```

CBSPTreeLoader::Build

The 'Build' function in the 'CBSPTreeLoader' class is a virtual function declared initially by the base 'ISpatialTree' interface. In each of the tree classes we have developed to date, the application has used this function to trigger the actual tree compilation process after the scene data has been added. However,

as we mentioned earlier, this class will not actually implement any kind of compile behavior. At this point, the information associated with the tree has already been compiled by our pre-processing tool and written to file. As a result, the 'Build' function of this class is primarily responsible only for setting up and calling the various import and reconstruction procedures that will create and populate the appropriate tree data structures.

One thing that you may notice as we move through the code in this class is that it only imports a small portion of the source IWF file specified in the class constructor. At no point does this class ever load or manipulate any type of scene polygon data for instance. This is due to the fact that the application will already have imported much of this information on our behalf. With the polygon data in particular, these will have already been loaded from the file in the 'CScene' class, and added to the member arrays inherited by this class using the 'CBaseTree::AddPolygon' function. With the scene responsible for loading much of the physical level data, this class needs only to import and process the BSP tree information that was exported by the tool developed in lab project 16.2. Any other information that may be stored in the file will simply be ignored. Thanks to the 'CBaseIWF' class exposed by the libIWF library, this is actually made very simple.

As defined by the 'ISpatialTree' super-class, the 'Build' method accepts a single parameter as input. This parameter is named 'bAllowSplits' and is a simple Boolean value that traditionally instructed the compiler as to whether the resulting polygon data should be split against the node planes or left whole. In the case of the 'CBSPTreeLoader' class, the tree and its associated polygon data have already been constructed and written to file based on the parameters specified in the pre-processing step. As a result this parameter will be ignored and the geometry will instead simply be used in its existing form.

```
bool CBSPTreeLoader::Build( bool bAllowSplits /* = true */ )
{
    try
    {
        // Open the file
        m_FileLoader.Open( m_strFileName, CBaseIWF::MODE_READ );

        // Set up the author ID for custom chunk reading
        m_FileLoader.SetAuthorID( "BSPC1", 5 );
```

Earlier we mentioned the 'm_FileLoader' member which is an instance of the 'CBaseIWF' class exported by the libIWF import library. This class handles the majority of the processing and navigating of an IWF file's structure and provides us with an easy means to gain access to the data stored within that file. In order to have this object process a specific file, the first thing we must do in this function is to open the file that we are interested in importing. This is achieved by calling the 'Open' method of the 'm_FileLoader' object passing in the name of the file to be processed. Recall that when an instance of this 'CBSPTreeLoader' class is created, the name of the IWF file that we are loading is passed to the constructor and duplicated into the 'm_strFileName' member variable. Therefore, this is the string that is passed as the first argument to the 'Open' method of the 'm_FileLoader' object. In addition, we must also instruct this object that the file should be opened for the purpose of *reading* data rather than writing. To this end, we also pass the 'CBaseIWF::MODE_READ' enumerator item as the second argument to this same function.

During the call to the ‘Open’ method, the file will be tested to ensure that it is of a format supported by the import library. Should this be the case, any appropriate file header information will be processed automatically and the ‘m_FileLoader’ object will be initialized such that we can begin reading data immediately. If an error occurred – such as the file being of an unrecognized format – the ‘CBaseIWF’ class will throw an exception that must be handled by our application such that we can return a failure code and potentially exit from the application if necessary. For this reason, the entire file loading logic is wrapped in a ‘try’ block that is used to this end.

In addition to specifying the source file that we would like this object to process, we must also inform the file object of the information that identifies any custom chunk data contained in the file as belonging to this application. Recall from our earlier discussions of the author ID with respect to entities and other custom data stored with the IWF, each of these items have a configurable signature that can be specified by the exporting application. This is done to ensure that only the application(s) that are specifically aware of its format ever try to import it. When loading custom chunks from the IWF file using the ‘CBaseIWF’ class, we can request that the object notifies us of these custom data chunks by specifying this signature with a call to the ‘SetAuthorID’ method of the ‘CBaseIWF’ class. This method accepts two parameters that require an arbitrary byte or character array – which denotes the author signature to verify against – and its associated length in bytes. Assuming that we pass in the same combination of signature bytes to this function as was specified during export, the IWF processing object will step into these custom data areas and correctly process the information instead of ignoring it.

In order to inform the ‘CBaseIWF’ object of the types of custom chunk we are interested in, we make use of the provided ‘RegisterChunkProc’ method. This function accepts the chunk ID value as the first of its three parameters. Recall that when exporting the plane, node and leaf data in the pre-processing tool we used the following three defines to identify each type of custom chunk written to file:

```
#define CHUNK_CTM_PLANES    0x2000
#define CHUNK_CTM_NODES     0x2001
#define CHUNK_CTM_LEAVES    0x2002
```

By re-using these same identifier values we are instructing the IWF processing object that we are interested in being notified whenever these three chunk types are encountered within the file. The means by which the IWF library notifies us of these occurances is by using the same callback mechanism we have used many times before in our lab project framework. By passing a function pointer to the second parameter of the ‘RegisterChunkProc’ method for each type of chunk, those static callback functions will be executed after the file has been positioned at the start of the appropriate data area.

As we know, due to the fact that these types of callback functions are static, it is also useful for the calling function to specify a context pointer. This is often a piece of custom data that can be used to pass any required information on to the callback procedure. In this application we pass in the pointer to the ‘CBSPTreeLoader’ class instance which is currently being processed. This context value is passed to the third parameter of this function to which we simply specify the ‘this’ pointer in each case. In doing so we allow each callback function that is executed to access the members of this specific tree object such that it might store any information that may have been loaded.

```
        // Load leaf and node data
        m_FileLoader.RegisterChunkProc( CHUNK_CTM_PLANES, ReadBSPPlanes, this );
```

```
        m_FileLoader.RegisterChunkProc( CHUNK_CTM_NODES , ReadBSPNodes, this );
        m_FileLoader.RegisterChunkProc( CHUNK_CTM_LEAVES, ReadBSPLeaves, this );
```

The callback functions that we register with the file loader object must be static and should each have a function signature that matches exactly with the one expected by this class. The function pointer typedef for the chunk callback procedures is shown below.

```
typedef void (*CHUNKPROC)(LPVOID pContext,CHUNKHEADER& Header,LPVOID pCustomData);
```

As we can see, each callback procedure will be passed three parameters and is not expected to provide any sort of return value. Given this specification we might imagine any given chunk procedure to be declared similar to the following:

```
static void MyProc( LPVOID pContext, CHUNKHEADER& Header, LPVOID pCustomData );
```

We will shortly discuss the specific details of each of the chunk callback functions that we registered here and will discuss the function parameters and their purpose in each case.

Now that we have registered the callback functions for each of the custom chunk types we would like to be informed of we can instruct the ‘m_FileLoader’ object to begin the processing of the currently open file. This is achieved by calling the ‘ProcessIWF’ method which will proceed to step through the various chunks in the file and call the appropriate procedure should any registered chunk type be encountered. Once the file has been processed in its entirety by this method and the registered callback procedures, we can then close the open file handle with a simple call to the ‘m_FileLoader.Close()’ method.

```
        // Load the additional BSP specific data
        m_FileLoader.ProcessIWF( );

        // Close the file
        m_FileLoader.Close();

    } // End Try Block

    catch ( ... )
    {
        // Complete Failure
        return false;

    } // End Catch Block
```

At this point, the registered chunk procedures should have loaded any node, leaf and plane data required for us to reconstruct the BSP tree. If there was no relevant tree data found to exist in the file, or a problem occurred for whatever reason, we should not continue with the reconstruction process. To this end we must verify that the tree data was indeed loaded by checking both the value contained in the ‘m_nFileNodeCount’ variable, in addition to the size of the ‘m_Leaves’ STL vector inherited from the base ‘CBaseTree’ class. If either of these members indicates that no relevant data was loaded then we return a failure code back to the calling function. Again, we will see the importance of checking these two variables specifically when we move on to discuss the actual chunk procedures registered earlier.

Should the required data have been imported successfully, we are now able to put together the various components to construct the spatial hierarchy. This is achieved with a call to the 'BuildTree' function which is a method of this class. We have encountered this function several times in previous lessons whereby it was traditionally responsible for the actual compilation and processing of the scene data in order to compile a particular type of spatial hierarchy. With the information having already been loaded from file, this function will simply be responsible for taking that information and rebuilding the tree structure in a compatible format. We will examine this function in a little while, but for now all that is important at this stage are the arguments that are being passed in to this function.

```
    // No BSP tree data loaded?
    if ( m_nFileNodeCount == 0 || m_Leaves.size() == 0 ) return false;

    // Reconstruct the tree structure from that loaded
    BuildTree( &m_pFileNodes[0], NULL );
```

The first parameter declared by the 'BuildTree' function is the current *source* node of the type 'iwfNode' (our temporary file based structure). Due to the fact that the node data is currently stored in a flat linear array, each of which contain only indices to inform us of the various parent/child relationships, it might not be instantly apparent exactly where to begin with the reconstruction process. Recall from our earlier coverage of the centralized node array in lab project 16.2 however, we always allocated the root node such that it would be located in the first element of that array. Therefore, using the temporary node information we loaded from the source IWF file, we begin this process by passing in that first element to the 'BuildTree' function. The second parameter is the current *destination* node that we will be constructing from the loaded source data during the recursive process. Since we have no root node at this point we simply pass NULL to this first call of the 'BuildTree' method. We will see how this information is used later in this lesson.

During the import of the spatial hierarchy data, recall that we used temporary arrays in order to store the node and plane information ready for processing. Now that we have fully reconstructed the BSP tree, this information is no longer required. As a result, we call a new method of this class called 'ReleaseFileData'. This function simply releases any memory associated with those temporary arrays to ensure that we don't consume any more memory than is necessary during the lifetime of the application.

```
    // Release the file data we had loaded.
    ReleaseFileData();

    // Allow our base class to finish any remaining processing
    return CBaseTree::PostBuild( );
}
```

At this stage we have imported, built and populated the arrays and hierarchy structures for each of the node and leaf items required. This is all of the information we need in order to begin using the BSP tree we compiled separately. With all of the pre-requisite information now available, we finally call the 'PostBuild' function of the base tree class to allow it to perform any additional processing on the tree data we have built. Recall that the 'PostBuild' procedure is responsible for constructing the vertex and index buffers that will be used during the rendering of the scene. The result of the 'PostBuild' function is returned directly to the calling function in order to signify the final success or failure of the build operation.

With the primary build interface function now covered, let us move on to take a look at the chunk procedure callbacks we registered in order to allow us to load the relevant components of the BSP tree hierarchy.

CBSPTreeLoader::ReadBSPPlanes

The first registered chunk procedure that we come to is the 'ReadBSPPlanes' function. As with all such registered procedures this function is defined as being static and must conform precisely to the function signature we discussed in the coverage of the 'Build' function earlier. This callback function accepts three parameters. The first of these is a void pointer that is used to inform each callback function of the context in which it is being executed. We have seen this type of callback system used many times throughout previous lessons, so the concept of a callback context should be nothing new to us. When registering this callback function with the 'CBaseIWF' object in the 'Build' function, recall that we passed a value of 'this' to the context parameter of the 'RegisterChunkProc' method. This of course means that the underlying object to which this first parameter will point is an instance of the 'CBSPTreeLoader' class that we are currently in the process of initializing. In this function, the loader class instance referenced by the context parameter will be the destination for any plane data that may be encountered in this part of the file. It is generally a good idea to cast this pointer to the expected type early on in the callback function to ensure that the code remains as simple and clean as possible.

The second and third parameters are not used by this application and are reserved mostly for use by the internal functionality of the 'CFileIWF' class exported by the libIWF library. As a result we will not spend too much time discussing their purpose. Put simply however, these parameters are used to pass additional file chunk information to the various callback procedures registered with the system. The 'Header' parameter for instance references a structure that contains various chunk properties such as the type of chunk being processed and the length in bytes of the chunk data area. For more information on the various pieces of additional chunk information available here, please refer to the IWF specification and SDK which should be available to you in your class supplemental download area.

At the point at which any registered chunk procedure is called by the IWF processing class, the file being processed should be positioned at the start of the relevant chunk data area within the file. What this essentially means is that we can begin reading the data relevant to this particular type of chunk immediately on entering this function. In order to do so however, we must gain access to the 'CBaseIWF' object instance that is being used to load the tree information. At the start of this function therefore, we cast the specified context pointer to that of our 'CBSPTreeLoader' class instance that we are populating, and then retrieve from that object a pointer to the 'm_FileLoader' member that specifies the 'CBaseIWF' object for the file we are currently reading from.

```
void CBSPTreeLoader::ReadBSPPlanes( LPVOID pContext, CHUNKHEADER& Header,
                                    LPVOID pCustomData )
{
    ULONG i;

    // Retrieve context pieces.
    CBSPTreeLoader * pLoader = (CBSPTreeLoader*)pContext;
    CBaseIWF *pFile = &pLoader->m_FileLoader;
```

Now that we have access to both the source file object and the destination 'CBSPTreeLoader' class instance, we can begin to load the plane data from the file. We achieve this directly through the file / stream handling methods exposed by the 'CBaseIWF' class which include functions such as 'Read', 'Write' and 'Seek'. A full list of each of these methods and their usage can be found in appendix A of this chapter's workbook. The first item we come to is a simple four byte 'unsigned long' that specifies how many planes – in total – are stored within the file. We read this value with a call to the 'CBaseIWF::Read' method, passing in a pointer to the 'm_nFilePlaneCount' member of the 'pLoader' object into which the plane count will be loaded directly, bypassing the need for a temporary variable.

Next on the agenda is to allocate enough room in our loader class' temporary 'm_pFilePlanes' array to store every plane contained within this chunk. The number of planes is of course described by the 'm_nFilePlaneCount' variable we initialized prior to this step and is used to correctly size the array with the new operator. Before we move on, this new array is cleared with a call to the Win32 'ZeroMemory' function such that we begin with a sensible set of default values.

```
    // Retrieve the Plane Count
    pFile->Read( &pLoader->m_nFilePlaneCount, sizeof(ULONG) );

    // Allocate storage for the planes
    pLoader->m_pFilePlanes = new D3DXPLANE[ pLoader->m_nFilePlaneCount ];
    ZeroMemory( pLoader->m_pFilePlanes,
                pLoader->m_nFilePlaneCount * sizeof(D3DXPLANE) );
```

With the temporary plane array allocated, the actual plane information can now be loaded and stored ready for the reconstruction step which occurs once the file has been completely processed. The code used to achieve this is shown below in which a loop is created that iterates through each of the planes in the new temporary plane array. These planes are used as the destination into which each subsequent plane contained within this file chunk is loaded and stored. Similar to loading the simple unsigned long value we encountered earlier, this is achieved with a call to the 'CBaseIWF::Read' method passing in a pointer to the destination plane structure stored in the temporary array. In addition we also pass a value which indicates the number of bytes that must be read from the file in order to load the plane in its entirety. This process is repeated for each plane indicated by the 'm_nFilePlaneCount' member until all planes have been loaded from the file and stored in the temporary 'm_pFilePlanes' member array.

```
    // Read Planes
    for ( i = 0; i < pLoader->m_nFilePlaneCount; i++ )
    {
        D3DXPLANE * pPlane = &pLoader->m_pFilePlanes[i];

        // Load the plane information from file
        pFile->Read( pPlane, sizeof(D3DXPLANE) );

    } // Next Plane
}
```

At this stage, the 'm_pFilePlanes' array contained within the 'CBSPTreeLoader' class has been allocated and fully populated with that data exported by the pre-processing tool developed in lab-project 16.2. The plane data is left in its combined form to ensure that the plane indices maintained by each

node remain valid. Of course, in order for the plane information to be of any use to us, we must also import the node data. Therefore, let us now examine the chunk callback procedure responsible for loading that information.

CBSPTreeLoader::ReadBSPNodes

As we observed with the 'ReadBSPPlanes' function, the static callbacks we must implement in order to load the custom tree data from the IWF file are relatively straight forward. The same is true when loading the node data into our temporary array of 'iwfNode' structures.

The majority of the code in this function is identical to that of the plane loading callback we discussed previously so we will not spend much time discussing the details here. This function simply reads the value describing the total number of nodes into the 'm_nFileNodeCount' member variable and allocates the temporary node array in much the same way as we did in the 'ReadBSPPlanes' callback. Due to the fact that we have defined the interim 'iwfNode' structure using the same format and layout of the node items contained in the file, we then simply read the node data one item at a time storing that information directly into each element in that temporary node array.

```
void CBSPTreeLoader::ReadBSPNodes( LPVOID pContext, CHUNKHEADER& Header,
                                   LPVOID pCustomData )
{
    ULONG i;

    // Retrieve context pieces.
    CBSPTreeLoader * pLoader = (CBSPTreeLoader*)pContext;
    CBaseIWF *pFile = &pLoader->m_FileLoader;

    // Retrieve the Node Count
    pFile->Read( &pLoader->m_nFileNodeCount, sizeof(ULONG) );

    // Allocate storage for the nodes
    pLoader->m_pFileNodes = new iwfNode[ pLoader->m_nFileNodeCount ];
    ZeroMemory( pLoader->m_pFileNodes,
                pLoader->m_nFileNodeCount * sizeof(iwfNode) );

    // Read Nodes
    for ( i = 0; i < pLoader->m_nFileNodeCount; i++ )
        pFile->Read( &pLoader->m_pFileNodes[i], sizeof(iwfNode) );

}
```

CBSPTreeLoader::ReadBSPLeaves

The 'ReadBSPLeaves' callback is by far the most complex of the tree import procedures implemented here. This is primarily due to the fact that there is much more information to be loaded, processed and stored in each leaf than in either of the other two cases. Similar to the other two callbacks defined within this class, this method is responsible for loading the leaf data from file. However, unlike those procedures, this method will load and populate an instance of a 'CBSPTreeLeaf' object rather than a temporary data structure. As we know, the reason we loaded the node and plane data into temporary arrays was due to the dependency that exists between them and the fact that we could not be certain

which of the two data chunks would be encountered first within the IWF file. In the case of the leaf however, all pre-requisite information has already been loaded by the scene import process, or is contained directly within the file leaf chunk we are reading from in this function.

As with each of the previous two functions, this is a static callback that accepts a void context pointer along with any appropriate chunk and custom data information neither of which is used by this application. Again, we cast the specified context pointer to that of our 'CBSPTreeLoader' class instance that we are populating, and retrieve the 'm_FileLoader' member that specifies the 'CBaseIWF' object for the file we are currently processing.

```
void CBSPTreeLoader::ReadBSPLeaves( LPVOID pContext, CHUNKHEADER& Header,
                                    LPVOID pCustomData )
{
    ULONG       i, j, LeafCount, PolygonCount, PortalCount, ReservedCount;
    D3DXVECTOR3 vecMin, vecMax;
    long        PolygonIndex;

    // Retrieve context pieces.
    CBSPTreeLoader * pLoader = (CBSPTreeLoader*)pContext;
    CBaseIWF *pFile = &pLoader->m_FileLoader;
```

Now that we have access to the source and destination objects we can begin to read the leaf data from file. The first item we come to is a simple four byte 'unsigned long' that specifies how many leaves – in total – are stored within the file. We read this value with a call to the 'CBaseIWF::Read' method, passing in a pointer to the local 'LeafCount' variable declared at the top of this function. We don't need to store this information directly as the tree class's internal leaf count will be incrememented as each leaf is finally added. If no leaves are found to be contained within the file we simply return.

Now we have the information that describes the total number of leaves in the tree, we can reserve the 'm_Leaves' member STL vector of the tree object using the vector's 'reserve' method. This will prevent any subsequent 'CBaseTree::AddLeaf' call from having to resize / reallocate the leaf array.

```
    // Retrieve the Leaf Count
    pFile->Read( &LeafCount, sizeof(ULONG) );
    if ( LeafCount == 0 ) return;

    // Allocate storage for the leaves
    pLoader->m_Leaves.reserve( LeafCount );
```

With this step completed, we can now begin to read the leaf information from file. The 'CHUNK_CTM_LEAVES' chunk stores all of the leaves in the tree together in one linear block. The size of this block of leaves is indicated by the 'LeafCount' variable that was read from the file a moment ago. At this point therefore we create a loop that will execute for this same number of iterations as each leaf is loaded.

The first thing we must do within this loop is to allocate a new 'CBSPTreeLeaf' object that will eventually be inserted at some point into the tree hierarchy. This leaf object will be used to store the information read directly from the file rather than into a temporary structure as with the node type. Remember from our earlier discussion that each leaf must have access to a pointer of the tree object in

which it is contained. This is done in the following code – in a similar manner to the previous compilation classes – by passing the tree object's pointer into the single parameter of the 'CBSPTreeLeaf' constructor. Unlike previous applications however, the 'ReadBSPLeaves' function is static and, as a result, we do not have access to the 'this' pointer. If you refer back to the earlier discussion of the 'Build' function, you will remember that when we registered each callback the 'this' pointer was specified as the callback context at that time. At the start of this function we cast the context pointer to the correct tree class type and copied it over into the 'pLoader' variable. Therefore, we can instead simply pass in the 'pLoader' variable to the leaf's constructor.

What follows this allocation is the reading of the various pieces of leaf data contained in the file. The majority of this information is loaded into temporary variables due to the fact that the actual leaf properties will be updated as we perform the various operations such as adding polygon data or setting the leaf's bounding box via the 'AddPolygon' or 'SetBoundingBox' methods inherited from the 'CBaseLeaf' class. The first two values to be loaded here are the minimum and maximum bounding box vectors that describe the absolute world space extents of the polygon data contained in this leaf. These are stored in the local 'vecMin' and 'vecMax' variables. Following these vectors is a value that is currently reserved for a later lab project. This is a single unsigned long that will eventually be used to store an index into the main visibility array that we will be constructing in the next lesson. For now we simply seek over this value.

The next two values contained in the file describe the total number of elements for both the polygon and portal index data that we will shortly encounter in the file. Again, the 'PortalCount' variable is not used by this application and is reserved for the visibility compiler we will build in the next lesson. However, it is important that we read this information in case we attempt to import a file built by the later version of our pre-processing tool such that we can skip over any portal data that may be contained in this file for this leaf. We will discuss the means by which this is achieved a little further on, but for now just know that these two variables describe the total number of indices maintained by the leaf for both the polygons and portals that are attached to this leaf.

The final of these count variables is the 'ReservedCount' item. Again this is not used by any of our current lab projects, but this reserved concept can be used by an application to store custom information along with the leaf data. The value retrieved by this read operation will describe the amount of space reserved for this custom data that will allow us to skip that portion of the leaf should it not be of any interest to us. Again we will talk about this more specifically further on in this function's coverage.

```
    // Read Leaves
    for ( i = 0; i < LeafCount; i++ )
    {
        CBSPTreeLeaf * pNewLeaf = new CBSPTreeLeaf( pLoader );

        // Read initial part of structure
        pFile->Read( &vecMin, sizeof(D3DXVECTOR3) );
        pFile->Read( &vecMax, sizeof(D3DXVECTOR3) );
        pFile->Seek( sizeof(ULONG) ); // Seek over PVSIndex
        pFile->Read( &PolygonCount, sizeof(ULONG) );
        pFile->Read( &PortalCount, sizeof(ULONG) );
        pFile->Read( &ReservedCount, sizeof(ULONG) );
```

With this first part of the leaf now loaded and stored we can begin to initialize the actual leaf object we created in the initial stages of this loop. The first thing we do here is to set the leaf's bounding box using the local 'vecMin' and 'vecMax' variables that were populated in the first two read operations for this leaf. Once this has been done we then begin to link any appropriate polygon data to the leaf object itself. To do this, we create a loop that will execute for the total number of iterations specified by the value stored in the 'PolygonCount' variable we loaded earlier. Recall that this value describes the total number of indices stored in this particular leaf that indicate which polygons we should attach here. At each step we then read in a single 4 byte value from the file that describes the index of the polygon to be attached to this leaf. Due to the fact that the polygon data has been loaded by the scene class and added to the 'CBSPTreeLoader' polygon array in advance we can simply retrieve the pointer stored at that element in the inherited 'm_Polygons' array, and pass that pointer directly into the new leaf's 'AddPolygon' method. This will proceed to take that pointer and update the leaf's internal polygon list to store the specified polygon.

```
        // Store bounding box details in the new leaf
        pNewLeaf->SetBoundingBox( vecMin, vecMax );

        // Read the face indices
        for ( j = 0; j < PolygonCount; ++j )
        {
            // Read the index into the polygon array
            pFile->Read( &PolygonIndex, sizeof(long) );
            if ( PolygonIndex < 0 ) continue;

            // Add the specified polygon to the leaf
            pNewLeaf->AddPolygon( pLoader->m_Polygons[ PolygonIndex ] );

        } // Next Leaf
```

Once this loop has completed the required number of iterations, we should have loaded each of the polygon indices contained in the file for this leaf, and added the physical polygon pointers to the new leaf object that we are building. With this operation completed, we can now move on to process the remaining items stored in the file.

Due to the fact that we are not interested in any exported portal index information in this application, the final thing we must do is to skip over any such information that *may* be contained in the file. Although the pre-processing tool developed in lab project 16.2 does not export these leaf portal indices, it is advisable that we do this regardless in case the source IWF file was not exported by that version of the compiler. Notice below that we also seek over the actual reserved data area. The size of this area is determined by the value read and stored in the local 'ReservedCount' variable which describes the number of DWORD sized chunks (32 bits / 4 bytes) of reserved data that is stored here. Again, no reserved data is written to file in lab project 16.2 and the 'ReservedCount' variable should contain a value of 0. However, as with the portal indices, we seek over any data that may have been indicated.

Now that we have read all of the data for this leaf, we can add it to the tree. However, we do not yet have access to the full spatial hierarchy and are unable to attach this leaf to any node that might reference it. As a result, we simply pass the new leaf into a call to the tree class' 'AddLeaf' method – inherited from the 'CBaseTree' class – that will proceed to store the leaf in the main centralized array. We can later retrieve this pointer when we reconstruct the tree hierarchy in the 'BuildTree' function.

```
        // Skip the reserved data areas
        pFile->Seek( PortalCount * sizeof(long) );
        pFile->Seek( ReservedCount * sizeof(long) );

        // Add this leaf to our leaf array
        pLoader->AddLeaf( pNewLeaf );

    } // Next Leaf
}
```

At this stage, the file should be positioned at the start of the *next* leaf ready to be loaded in the next iteration of this loop. We continue this entire process until all of the indicated leaves have been loaded and stored in the tree's centralized leaf array.

Once this loop has read and stored the leaves contained within the file and, assuming that the node and plane data has already been loaded at this stage, we now have all the information we need to be able to reconstruct the tree. Let us now take a look at how this is achieved.

CBSPTreeLoader::BuildTree

There have been several references to the reconstruction of the BSP tree throughout our coverage of the lab project 16.3 implementation. In previous applications, we developed a private 'BuildTree' function which served as the main recursive procedure in which the tree construction was undertaken. In lab project 16.3, this function has a similar role in that it must recursively build the spatial hierarchy. However, this time it is solely responsible for taking any information that has been previously loaded by the 'Build' function and piece it together such that it now exists in the appropriate format for use by our run-time rendering application.

This function accepts two parameters. The first of these indicated by the 'pFileNode' parameter is the current interim file node that will be used as the source for building a 'CBSPTreeNode' that will be used in the final spatial hierarchy for this tree object. The second 'pNode' parameter is the destination 'CBSPTreeNode' object that is to be populated with any relevant information. Recall from the earlier discussion of the 'Build' function that the first time that this 'BuildTree' method is called, a value of NULL is passed to the 'pNode' parameter. This is used to indicate that we are constructing the root node of our final spatial hierarchy. As a result, the first task undertaken by this function is to allocate a new root node and store the resulting pointer in the tree's 'm_pRootNode' member variable. Of course, this will only happen the first time that this method is invoked due to each subsequent call being passed an already instantiated destination node object.

After creating the root node, or simply using the destination node that was passed, we begin populating this node with the simple bounding box and plane information indicated by the interim 'iwfNode' structure passed to this iteration of the recursive procedure. The first of these is the separating plane on which the node lies. Recall that we imported the combined plane array from the file and stored these in a temporary 'm_pFilePlanes' member array. Due to the fact that the final 'CBSPTreeNode' class stores a physical plane item, the plane indicated by the file node must be extracted from this array and stored directly into the destination node object. The final two values we update in the initial stages of this

function are the ‘BoundsMin’ and ‘BoundsMax’ values which are taken directly from the imported node structure.

```
bool CBSPTreeLoader::BuildTree( iwfNode * pFileNode,
                                CBSPTreeNode * pNode /* = NULL */ )
{
    D3DXVECTOR3 vecMin, vecMax;

    // First time in?
    if ( !pNode )
    {
        // Allocate a root node
        if ( !(pNode = new CBSPTreeNode) ) return false;
        m_pRootNode = pNode;

    } // End if no node specified

    // Build node data from that loaded from file
    pNode->Plane     = m_pFilePlanes[ pFileNode->PlaneIndex ];
    pNode->BoundsMin = pFileNode->BoundsMin;
    pNode->BoundsMax = pFileNode->BoundsMax;
```

With these simple properties updated, this function now proceeds to process and create the child node and leaf items where necessary. The first case we come to is the creation of any applicable front child. Because we are always guaranteed to have a front child whenever we are dealing with a node that does **not** store a leaf, we first allocate a new ‘CBSPTreeNode’ item which will represent the child in front of the current node. The pointer to this new node item is then stored into the ‘Front’ member of the node we are currently processing. Once we have allocated this new front node item, this function then tests to determine whether the interim file node indicates that a child *node* or child *leaf* is attached to the front side of the node at this level. The first case we come to is one in which a child *node* has been indicated by the ‘FrontIndex’ member of the imported node structure, in which we would find a positive index value. If this turns out to be the case, then there is nothing further for us to do with the current node and we simply recurse into the new front child node by making a further call to the ‘BuildTree’ function. At this stage we pass in as the first of two parameters the imported node that was specified by the ‘FrontIndex’ value of the current ‘iwfNode’, as well the node destination node we allocated a moment ago. Should we enter this initial ‘if’ statement, then this process would begin again such that the entire subtree of the current node is built and attached to the front side of the current destination ‘CBSPTreeNode’ object.

```
    // Allocate new node in front
    pNode->Front = new CBSPTreeNode;
    if ( !pNode->Front ) return false;

    // Node or leaf in front?
    if ( pFileNode->FrontIndex >= 0 )
    {
        // Build this new front node
        if ( !BuildTree( &m_pFileNodes[ pFileNode->FrontIndex ], pNode->Front ) )
            return false;

    } // End if node in front
```

The alternate case that we have to deal with is one in which there is a child *leaf* attached to the front side of the current node. This is indicated by the 'FrontIndex' value of the imported node having a **negative** value. Because the initial 'if' statement simply test for any positive value including zero, we can ensure that we are dealing with a leaf simply by using the 'else' keyword. If we drop into this 'else' case we must of course attach a 'CBSPTreeLeaf' item to the front of this node. However, recall that our run-time spatial hierarchy design requires that a leaf-node always be inserted above any given leaf in the tree. Due to the fact that we always explicitly allocated a front node earlier in this function, this is taken care of automatically. By simply attaching the indicated leaf to the **font child** rather than the current node, we can always ensure that this parent leaf-node exists. Notice that we first retrieve the pointer to the correct leaf from the tree's existing leaf array using the leaf indexing logic we discussed in lab project 16.2. Recall that because the value for any indices available for describing a leaf in the file's front or back index members is in the range of -1 and below we must add 1 to the 'FrontIndex' member here before flipping its sign. By doing so this value is converted back into a valid array index in the range of 0 and above.

```
    else
    {
        // Build a leaf
        CBSPTreeLeaf * pLeaf =
                       (CBSPTreeLeaf*)m_Leaves[ abs(pFileNode->FrontIndex + 1) ];
        if ( !pLeaf ) return false;

        // Store pointer to leaf in the node
        pNode->Front->Leaf = pLeaf;

        // Store the leaf's bounding box in the node
        pLeaf->GetBoundingBox( vecMin, vecMax );
        pNode->Front->BoundsMin = vecMin;
        pNode->Front->BoundsMax = vecMax;

    } // End if leaf in front
```

With the front side of this node and the entire front subtree fully reconstructed, this function now turns its attention to the back. We perform the initial steps in an identical fashion to that of the front simply substituting each of the relvant front indices and pointer variables with those specifying the information attached to the back of each node. In the same way as before, we first allocated a new back child 'CBSPTreeNode' object for the current node and then recurse into the back subtree should the value of the 'BackIndex' member of the imported node indicate that a child *node* should be attached here.

```
    // Allocate new node behind
    pNode->Back = new CBSPTreeNode;
    if ( !pNode->Back ) return false;

    // Node or leaf behind?
    if ( pFileNode->BackIndex >= 0 )
    {
        // Build this new back node
        if ( !BuildTree( &m_pFileNodes[ pFileNode->BackIndex ], pNode->Back ) )
            return false;
```

```
    } // End if node in front
```

Once again, the alternate case is one in which an attached child *leaf* is indicated by the imported node data. In this case, we perform exactly the same steps as discussed previously. However, there is an additional consideration that must be taken with this type of compiled BSP tree information when processing the information for the back side of any given node. This is of course when the back halfspace of that node is describing an area of solid space.

If we think back to the development of the BSP leaf tree compilation process, solid leaves were not physically created or inserted into the tree and instead a value equal to that of the 'BSP_SOLID_LEAF' defined constant was stored in the 'BackIndex' member of the node. To this end, the code within this else clause first tests to see whether the current file node's 'BackIndex' member contains this value. If it does then of course we must also signify this 'solid' area within our current spatial hierarchy framework. We achieve this by simply storing a value of 'NULL' in the back child of the current destination node after releasing the child node that we allocated earlier. We could alternately create a back leaf-node that simply stores a value of 'NULL' in its 'Leaf' pointer member. However, the former method that we have chosen to employ in this lab project is slightly more efficient in both processing and memory overhead and has no significant drawbacks.

If the space behind the node is not describing a solid area of space, then we simply attach the leaf indicated by the imported node in the same manner as with any front child leaf.

```
    else
    {
        // Solid leaf?
        if ( pFileNode->BackIndex == BSP_SOLID_LEAF )
        {
            delete pNode->Back;
            pNode->Back = NULL;

        } // End if solid leaf
        else
        {
            // Retrieve the leaf specified
            CBSPTreeLeaf * pLeaf =
                          (CBSPTreeLeaf*)m_Leaves[ abs(pFileNode->BackIndex + 1) ];
            if ( !pLeaf ) return false;

            // Store pointer to leaf in the node
            pNode->Back->Leaf = pLeaf;

            // Store the leaf's bounding box in the node
            pLeaf->GetBoundingBox( vecMin, vecMax );
            pNode->Back->BoundsMin = vecMin;
            pNode->Back->BoundsMax = vecMax;

        } // End if empty back leaf

    } // End if leaf behind

    // Success!
    return true;
```

```
}
```

At this stage, from the point of view of the first level of recursion, both the front and back subtrees will now have been fully reconstructed and attached to the front and back member pointers of the root node. Hopefully it should be clear that no physical compilation has really been undertaken here, we have simply taken the information stored in the file and converted it into the applicable format ready for our application to use.

CBSPTreeLoader::ReleaseFileData

Recall that the 'ReleaseFileData' method is called by the 'Build' function after the tree data has been loaded and reconstructed. Due to the fact that this information is only required during import – and would simply consume memory unnecessarily – each of these file data arrays should be released once the hierarchy has been built. This function is responsible for clearing these temporary file data arrays and resetting any appropriate values for the possibility of later use.

```
void CBSPTreeLoader::ReleaseFileData()
{
    // Destroy arrays
    if ( m_pFileNodes  ) delete []m_pFileNodes;
    if ( m_pFilePlanes ) delete []m_pFilePlanes;

    // Clear Variables
    m_pFileNodes      = NULL;
    m_pFilePlanes     = NULL;
    m_nFileNodeCount  = 0;
    m_nFilePlaneCount = 0;

}
```

CBSPTreeLoader::ProcessVisibility

In this method we begin to see the visibility call counting scheme from the point of view of the tree class. The 'ProcessVisibility' function is almost identical to those we have implemented in each of our spatial hierarchy classes so far. The only addition here is that we increment the 'm_nVisCounter' member of the *tree* class before we begin the recursive 'UpdateTreeVisibility' process. It is important that we do this first, because we need to ensure that each of those leaves subsequently found to be visible are updated with a value that will remain constant until the *next* call to this function. If this value was to be incremented after the update process, then the visibility counter value stored in each leaf will become out of date the moment we stepped out of the 'UpdateTreeVisibility' call.

```
void CBSPTreeLoader::ProcessVisibility( CCamera & Camera )
{
    CBaseTree::ProcessVisibility( Camera );

    // Increment the visibility counter for this loop
    m_nVisCounter++;

    // Start the traversal.
```

```
    UpdateTreeVisibility( m_pRootNode, Camera );
}
```

CBSPTreeLoader::UpdateTreeVisibility

Once again, the 'UpdateTreeVisibility' method defined by this new tree class is almost identical to those we have already encountered. However, there is one minor alteration to this function that has a significant impact from an execution point of view.

As we can see, with the exception of substituting the type of the 'pNode' parameter to that of our new node class, and including this method in the new 'CBSPTreeLoader' namespace, the initial part of this function remains unchanged. Each of the parameters are declared and used in the same manner as before, and the frustum culling result code is retrieved in the same way. Once we have retrieved this frustum value, we then come to the conditional 'switch' statement that is also part of each of the previous implementations.

```
void CBSPTreeLoader::UpdateTreeVisibility( CBSPTreeNode * pNode, CCamera & Camera,
                                           UCHAR FrustumBits /* = 0x0 */ )
{
    CCamera::FRUSTUM_COLLIDE Result =
                             Camera.BoundsInFrustum( pNode->BoundsMin,
                                                     pNode->BoundsMax, NULL,
                                                     &FrustumBits,
                                                     &pNode->LastFrustumPlane );

    // Test result of frustum collide
    switch ( Result )
    {
```

The first case we come to when determining if the current node's bounding box is fully within the frustum is that of the 'FRUSTUM_OUTSIDE' case. The only difference in this function can be found in here. Recall that this result is returned whenever that bounding box is *completely* outside of all of the camera's frustum planes and as a result cannot possibly seen. In addition we know that none of its children can possibly be seen either because of the fact that a node's bounding box will be large enough to contain every child node beneath it in that branch of the tree.

If you recall the earlier implementations of this function, we previously called the node's 'SetVisible' method, passing in a value of 'false' in this *outside* case. This function would then traverse this node's subtree simply in order to visit every node and leaf that existed as a child of the current node, marking each leaf encountered as invisible. With our new visibility call counter mechanism, this process of updating each leaf's visibility status when it is *not* visible is no longer required. As a result, this case now simply returns from the function without taking any additional action.

As we move forward and add concepts such as the 'Potential Visibility Set' – covered in the next lesson – this adjustment could save us a significant amount of additional processing in a highly occluded scene. However, even in the case of simple frustum culling, this technique can be employed in order for us to remove a large portion of the traversal operation we were previously required to perform during the 'ProcessVisibility' pass.

```
        case CCamera::FRUSTUM_OUTSIDE:
            // Node (and all its children) are not visible
            return;
```

The remaining cases in this switch statement remain unaltered from our previous implementations. In the ‘FRUSTUM_INSIDE’ case we must continue to recurse through the subtree of the current node in order to set each child leaf as visible. In the ‘FRUSTUM_INTERSECT’ case we must continue to perform the frustum tests for any child node until we find a node that is either contained within or is completely outside of the frustum. This is of course unless this is a leaf node, in which case the leaf must be set to a visible state to ensure that the visibility status information and rendering buffers are updated correctly for that leaf.

```
        case CCamera::FRUSTUM_INSIDE:
            // Node is totally visible
            pNode->SetVisible( true );
            return;

        case CCamera::FRUSTUM_INTERSECT:
            // We need to resolve this further, unless this is a leaf
            if ( pNode->Leaf )
            {
                pNode->SetVisible( true );
                return;
            } // End if leaf
            break;

    } // End Switch
```

As before, the final block of code is only ever executed if the bounding box of the current node was found to be intersecting the frustum planes. If this was the case then further consideration needs to be taken and as a result we must traverse down the front and back of this node.

```
    // The remaining case (FRUSTUM_INTERSECT) means we need to test further
    if ( pNode->Front ) UpdateTreeVisibility( pNode->Front, Camera, FrustumBits );
    if ( pNode->Back  ) UpdateTreeVisibility( pNode->Back, Camera, FrustumBits );
}
```

Additional CBSPTreeLoader Routines

There are several tree related support routines that we have developed in the past that allow the application to perform operations such as the collection of a list of leaves that are intersected by either an axis aligned bounding box or a ray, and the drawing of debug information. Due to the large similarities between the basic principals of both the BSP tree and the kD-tree, the ‘CBSPTreeLoader’ class duplicates several of these functions taken directly from the ‘CKDTree’ class. Because we have covered the kD-tree support methods in earlier lessons, we will not spend any time covering them again here.

The list below outlines those functions which are direct copies from the ‘CKDTree’ class that have simply been altered to be declared within the ‘CBSPTreeLoader’ namespace and also to make use of the new ‘CBSPTreeNode’ class.

```
bool CollectLeavesAABB    ( LeafList & List, const D3DXVECTOR3 & Min,
                            const D3DXVECTOR3 & Max );
bool CollectAABBRecurse   ( CBSPTreeNode * pNode, LeafList & List,
                            const D3DXVECTOR3 & Min, const D3DXVECTOR3 & Max,
                            bool bAutoCollect = false );
bool CollectLeavesRay     ( LeafList & List, const D3DXVECTOR3 & RayOrigin,
                            const D3DXVECTOR3 & Velocity );
bool CollectRayRecurse    ( CBSPTreeNode * pNode, LeafList & List,
                            const D3DXVECTOR3 & RayOrigin,
                            const D3DXVECTOR3 & Velocity );
void DebugDraw            ( CCamera & Camera );
bool DebugDrawRecurse     ( CBSPTreeNode * pNode, CCamera & Camera,
                            bool bRenderInLeaf );
bool GetSceneBounds       ( D3DXVECTOR3 & Min, D3DXVECTOR3 & Max );
```

Scene Class Modifications

With the new BSP tree loading class fully implemented, the only task that remains is to make any modifications to the application code such that this new spatial hierarchy class is utilized. Because of the way that the ‘ISpatialTree’ concept has been developed, switching to other types of tree within the application is trivial. We have observed on several occasions how we can alternate between an oct-tree, quad-tree and kD-tree simply by altering a single line in the ‘CScene’ class such that the relevant type of object is instantiated before the scene is loaded. The new ‘CBSPTreeLoader’ class is integrated into the application in exactly the same way. With this in mind let us look at the minor modification that we must make in the scene class’ ‘LoadSceneFromIWF’ function in order to make use of the compiled solid leaf BSP tree.

CScene::LoadSceneFromIWF

The following code block shows a small portion of the ‘LoadSceneFromIWF’ method that we should already be extremely familiar with. The only line of code that has changed here is marked in **bold**. We can see that when instantiating the spatial partitioning tree object we have simply created an object of the type ‘CBSPTreeLoader’, storing the returned pointer in the scene’s ‘m_pSpatialTree’ member variable. Recall that this member is of the type ‘ISpatialTree’ which allows the application to access the tree functionality without necessarily having to be aware of the specifics of the selected tree class itself.

When instantiating this type of spatial partitioning tree we pass in the common parameters required by all types of tree that we have developed. These are the Direct3D device that was registered with the

scene during application startup, along with the Boolean flag describing whether hardware transformation and lighting is being used. These two parameters are used by the underlying 'CBaseTree' class to create and populate the vertex and index buffers used in the rendering of the geometry registered with the tree. The third and final parameter is the one that is specific to the 'CBSPTreeLoader' class. Here we simply pass in the filename that was also passed to the 'LoadSceneFromIWF' method by the application. This means that both the scene geometry and the custom tree information will be loaded from the same file. Whilst this is convenient it is not a requirement. Due to the fact that the BSP tree loader class opens and processes the file independently of the scene class, it would be possible to have the scene geometry and BSP tree information stored in separate files. In this case we would simply pass in an alternate filename to the loader class constructor.

```
    ...
    ...

    // File loading may throw an exception
    try
    {
        // Allocate our spatial partitioning tree of the required type
        m_pSpatialTree = new CBSPTreeLoader( m_pD3DDevice, m_bHardwareTnL,
                                             strFileName );

        m_pAlphaTree   = new CBSPNodeTree( m_pD3DDevice, m_bHardwareTnL );

        // Add our scene callback to the player.
        GetGameApp()->GetPlayer()->AddPlayerCallback( CScene::UpdatePlayer, this );

        // Attempt to load the file
        File.Load( strFileName );

        ...
        ...
```

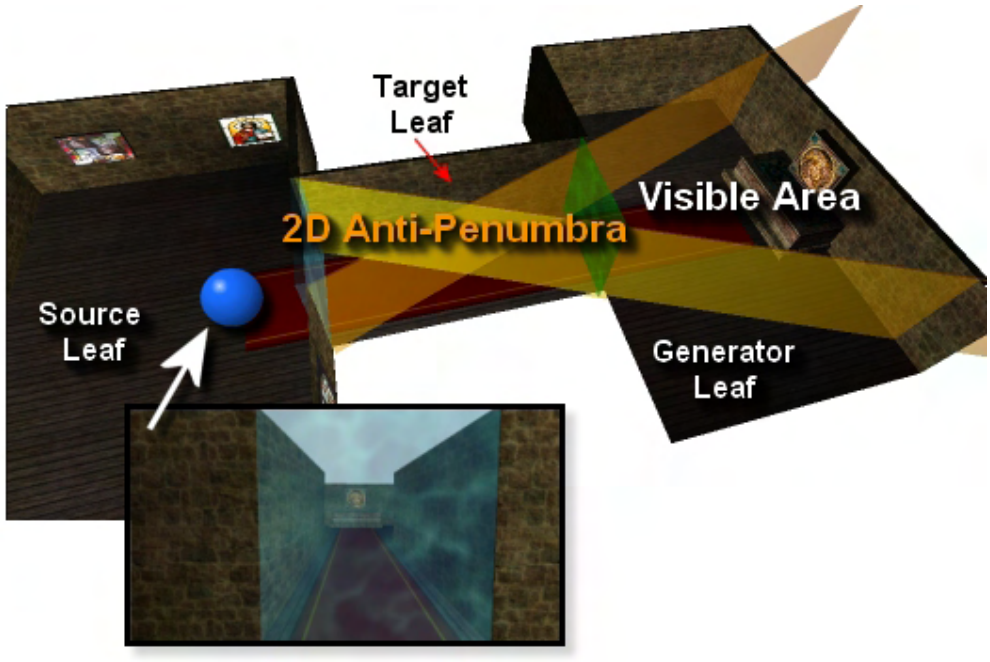
This line is all that needs to be altered in order for our application to begin using the compiled BSP tree information generated and exported by the pre-processing tool developed in lab project 16.2. It is still also entirely possible for the application to select an alternate spatial partitioning tree class at any point, using the same compiled IWF file. The polygon data that is loaded by the scene will simply be compiled as has always been the case in earlier lab projects.

Lab Project Conclusion

This concludes our coverage of the first BSP tree rendering application. In the next lesson we will enhance both the compiler tool and the BSP tree loader class to make use of accurate visibility information. This will be used to cull away any level geometry that cannot possibly be seen due to any other geometry that might be occluding the player's view of other parts of the scene. This should greatly enhance both the performance of our rendering application, but will also open up many opportunities for improving the efficiency of many tasks that we must undertake in the future when developing additional game technologies.

Chapter Seventeen

Spatial Partitioning IV



Introduction

In the previous lesson we discussed and implemented the BSP leaf tree which demonstrated yet another form of spatial partitioning. While one of the key use cases for the BSP node tree was made clear in Chapter 16, when we implemented a perfect alpha sorter, the ideal usage of the BSP leaf tree may still be a little unclear at this point. The BSP leaf tree certainly had some interesting characteristics. Like the node tree, it divided the level into arbitrarily shaped convex areas due to the fact that the split planes at each node were taken from the level geometry itself. Although useful, it might be hard to imagine at this time why this would somehow be preferable to our previously discussed bounding volume based spatial tree types. After all, the leaf tree generates many more nodes and leaves in the typical case and the bounding volumes assigned to each leaf are not tightly fitting like the quad-tree, oct-tree or kD-tree. This means that the BSP leaf tree would theoretically be slower to traverse and frustum cull in the typical case. The frustum culling applied to the leaves would also not be as precise as the other tree types due to the fact that the AABB we store in each leaf exhibited quite a loose fit. The result is that even axis aligned leaves that are outside the frustum might still get rendered because the loose fitting box intersects the frustum even though the geometry in that leaf does not. When viewed in this light, one might wonder in what situation the application of a BSP leaf tree would be a wise choice.

The key to understanding the usefulness of the BSP leaf tree is found in the properties it exhibits when fed geometry that follows the solid/empty guidelines. You will recall in the previous lesson that if we feed our BSP compiler geometry that has no hidden surfaces (i.e., the front of one polygon can never see the back of another), the tree provides us with the ability to distinguish between which areas of the level are assumed to define solid matter (such as the space behind a wall for example) and which areas are assumed to define empty space (the empty space between our polygons in which the player and NPCs are allowed to freely navigate). Providing geometry in a solid/empty form to the BSP tree compiler does mean that the game artist has to be especially vigilant when building the scene, although as we also learned that using CSG techniques to perform hidden surface removal as a pre-process before the compile step can remove hidden surfaces in the typical case. We saw in Lab Project 16.2 that we can run union operations on the individual brushes/meshes comprising the scene so that any faces of objects which are touching or intersecting each other are removed. This provides our BSP leaf compiler with a level that obeys the rules for what is considered *legal geometry* by a solid/empty leaf tree compiler. Even if the level was not created by the artist in a solid/empty format, as long as the individual meshes in the scene do not contain any faces that touch or intersect themselves, we can union these meshes into a single mesh (pre-compile) and remove any offending surfaces in the process. If the level we are given is a polygon soup (i.e., essentially a single mesh) however, or is comprised of meshes that individually contain illegal geometry, there is nothing we can do to resolve this situation and the compilation of a solid/empty leaf tree will not be possible.

Illegal geometry places a burden on the project artist to be extra careful during the level creation process, and it is for this reason that many artists use world editors (such as GILES™ or WorldCraft™) to build assets for use in a solid/empty environment. These editors allow the artist to build up a scene using a variety of primitives that can be carved, unioned or intersected with each other using CSG techniques to build more complex shapes that remain geometrically legal in the BSP leaf tree sense. For example, in GILES™ you can perform hidden surface removal on your final level before saving it out to disk by selecting all objects in the scene and then pressing the Union button. This action will perform a

union test between each brush with every other brush in the scene and discard any polygons that exist in solid space (such as those contained inside another brush). This will prepare the geometry for solid/empty BSP tree compilation.

Although the solid/empty property of the BSP leaf tree was an interesting one to discover, and certainly useful in the sense that it made our CSG routines possible, we will learn in this chapter that this information is actually going to be critical during the next stage of our engine development process. We will use it during the implementation of a new major module in our BSP leaf compiler tool -- the ability to calculate a Potential Visibility Set (PVS).

In this lesson we are going to learn how to leverage the power of BSP leaf trees to significantly reduce the number of polygons that have to be rendered each frame, allowing our applications to run much, much faster. Our new design will enable us to quickly reject unseen geometry, thereby separating the concept of frame rate from the size of the overall level. Using a BSP tree in combination with our Potential Visibility Set, we can reject nearly all geometry that is not currently visible with almost no processor or GPU overhead. This is because, like the leaf BSP tree in Lab Project 16.2, the Potential Visibility Set is pre-calculated at development time and saved out to disk along with the leaf tree data. Our renderer will no longer need to test each polygon or each leaf to see if it should be drawn. Instead, only polygons that can be seen from the current location within the scene are processed in any way.

Unlike our previous spatial tree demonstrations, the BSP leaf tree demonstration provided in the previous lesson was divided across two lab projects in order to separate out the compile process from that of the runtime renderer. The compiler itself was essentially implemented as a development tool that would compile the BSP tree and then save the data in its compiled format out to file. All the BSP tree information was included (the nodes, planes, leaves, polygons, etc.) so that all the renderer had to do was load in that data and use it to reconstruct the tree. This is obviously much quicker than waiting for the BSP tree to compile every time the application is run, especially on complex levels. Ideally we want our player to progress from one level to the next without long wait times in between. Of course, this is a good idea with any spatial tree partitioning scheme, so you could certainly use the previous lab project as a template for how you might save your oct-trees, quad-trees or kD-trees out to file. In each case, if the tree is static (not expected to change its form throughout the life of the application/level) it is generally better to pre-calculate this information at development time and package the level assets in their compiled form.

Although having to wait for an oct-tree compile (for example) can be an annoyance, the potential visibility set (PVS) calculator we add to our BSP leaf tree compiler is going to require much more time than a simple oct-tree compile. The calculation of PVS data is actually a very time consuming process. It can take anywhere from 5 minutes to several hours to complete, depending on the complexity and size of the scene data. Obviously this is not a process that we can perform in the final distribution of our game; it must be calculated at development time. Therefore, by separating out the leaf compiler into its own application in the previous lesson, we were actually laying the groundwork for a development tool that we will complete in this lesson. Lab Project 17.1 will be a BSP Leaf Tree Compiler and PVS Calculator that can be used at development time to compile our data and save it out to file. Lab Project 17.2 will demonstrate how to load and render the scene using the compiled BSP leaf tree and PVS data just generated. This highlights the actual rendering engine code which is shipped to the end user. Thus, at the end of this lesson we will have a tool to generate geometry files that are stored in BSP/PVS form and an application that demonstrates how a game engine can load and render this data.

17.1 Introducing Potential Visibility Sets

It is perhaps not overstating the point to say that the technology we will learn about and develop in this lesson is the most exciting we have learned thus far with respect to geometry management. It is even fair to say that this is the technology that we have been building up to over many of the previous lessons. The PVS calculator we develop in this chapter will open doors for us that were previously closed. It will allow us to render levels of such a size and complexity at frame rates that our engine, up to this point, could not possibly achieve. So what is a potential visibility set?

A Potential Visibility Set (PVS) is a set of visibility information stored for each leaf (generally in an array) which tells us which other leaves are considered visible from that particular leaf. We will look at how to generate and store PVS data a bit later when we cover the implementation of a PVS calculator, but for now, let us just understand the theory.

When covering the previous spatial tree types we discussed a visibility system that employed hierarchical frustum culling for efficient rejection of geometry outside the frustum. Yet, this is hardly what one might refer to a robust visibility system. In theory, a visibility system should tell you what is currently visible. Of course, our previous spatial trees did not exactly do this with much accuracy. They were quite good at telling us what was *not* in the view frustum (and thus not visible), but that did not lead to the conclusion that the geometry that was within the camera's view frustum was actually visible. Consider a camera located in a small room where, regardless of the size of the frustum, the only visible elements are the polygons comprising the floor, walls, and ceiling of that room. If we had a way of knowing that this was the case it would prevent us from having to render the possibly thousands of polygons located behind the polygons in the current room that are well within the FOV and far plane range of the frustum.

Because our previous visibility system did not take occlusion into account, in such a scenario, every polygon inside the frustum would be rendered, even if the camera was facing a wall and could see only that single polygon. Although the player would correctly see only that one polygon, the thousands that may be behind it still get drawn to the frame buffer during the render pass only to be eventually overwritten by the single polygon we can see. Even if the polygon we can see was, by luck, rendered to the frame buffer first, thus preventing any of the other polygons behind it from being rendered, those polygons would still need to be transformed, lit, and clipped by the pipeline only to be rejected at the per-pixel level during the depth test. This situation is what is commonly referred to as overdraw, and without implementing some form of occlusion culling in our engine, we might have a situation where, by the time we have rendered the final image into the frame buffer at the end of each frame, each pixel in the frame buffer may have been rendered and then replaced tens, if not hundreds, of times. This can be a serious problem when you consider that many of today's game players expect to play in very high pixel resolutions (e.g., up to 1600x1200). Even in a case where there is no overdraw whatsoever, the graphics hardware is going to have to paint 1,920,000 pixels every update (in a fraction of a second). If we imagine a scenario where each pixel has an average overdraw count of 25, that means the graphics hardware now has to paint 48,000,000 pixels to fill the same frame buffer. Even this one factor can lead to the difference between the game running smoothly and the game running at unacceptably low frame rates.

When one examines the origin of the overdraw problem, we naturally arrive at the conclusion that it is in highly occluded game environments where it is at its worst. A level of small passageways and rooms is one where we can imagine that thousands of polygons that lay behind the walls of the current hallway in which you reside are going to be rendered. A classic example is a game like Doom3™ or the original Quake™ series of games (or most other first person shooters) where the player is navigating through building interiors where the view is generally limited by surrounding geometry. The PVS calculator we develop in this lesson is tailored to take advantage of scenes like this that exhibit a high level of occlusion, and it will generate the visibility for each leaf taking that occlusion into account. For example, if two leaves were in very close proximity, but there was no line of site between these two leaves, they would both be omitted from each others visibility set.

> **Note:** The PVS system we will develop in this lesson does not work well with environments that have very little occlusion. A level comprised of a vast sprawling terrain for example is not a good candidate for this kind of PVS calculation. It is not that the PVS calculator will fail to work; simply that terrain occlusion requires a different (although not terribly dissimilar) approach. There are methods that can be implemented to calculate PVS for terrains, and you can find some good discussions of these techniques in the Game Programming Gems series by Charles River Media. Once you understand the basic PVS system that we develop in this chapter, you should have little trouble moving on to extend these ideas to other scenarios should you so choose.

Although we will discuss the implementation of a PVS calculator (with implementation specifics in the accompanying workbook) shortly, it should prove beneficial to start by discussing the data that will be produced by the PVS calculator and see how it will be used by our rendering application. This will give us a better idea of the data our rendering code will be working with to determine leaf visibility, along with a basic feel for the system as a whole. It will also identify the expected output of the PVS calculator.

The PVS calculator has a seemingly simple task. For each leaf in the BSP leaf tree, it will calculate a visibility set describing which other leaves should be rendered when the camera is in that leaf. We refer to the visibility information from all leaves as being the potential visibility set. That is, it contains potential visibility information for each leaf in the level, which taken together forms a set of potential visibility data for the whole scene. After this data has been saved out to file, the rendering application can load in that data and will have instant access, from any leaf, to the leaves it can see. As such, it knows immediately that the polygons in those leaves that should be rendered

For simplicity, let us just assume that each leaf in the tree maintains an array of bytes whose number of entries equals the total number of leaves in the tree. Imagine now that we are examining the PVS array for one of those leaves (call it “current leaf”). Each byte in this array is set to either 1 or 0 for every other leaf in the tree (assume that the leaves also have their pointers stored in a linear array or STL vector). This value basically tells us whether that leaf is visible from the current leaf. Byte 0 represents Leaf 0, byte 1 represents Leaf 1, and so on. For this example, assume that we are in Leaf 10. We loop through Leaf 10's PVS data and check byte 0. If this byte is set to 1 then Leaf 0 is visible from Leaf 10 and thus its polygon data should be collected into leaf bins and rendered; otherwise it should be skipped. You should see where this strategy is starting to lead us. By pre-calculating a PVS for each leaf ahead of time, not only can we avoid rendering unseen polygons, we will not have to process them at all. There will be no costly tests to see if polygons are within the view frustum because if they are not in that leaf’s PVS, then the leaf is simply skipped. The polygons in that leaf are never transformed, lit, culled or processed in any way. This also means that the only tree traversal we have to do is with the camera’s

location. We send it down the tree until it pops out in a leaf, fetch this leaf's PVS data array, and collect/render any polygons contained in leaves which have their PVS byte set to 1 in the current leaf's array.

In the next section we will examine what exactly the word 'potential' means with respect to a potential visibility set and talk about the amount of occlusion culling we can expect to get from such a system.

17.2 Potential Visibility

Figure 17.1 shows some example geometry that might be used in the calculation of a potential visibility set. We are looking at the level after the BSP tree has been compiled into eight leaves. For the purpose of explanation, we are assuming that each room in the level comprises a leaf, even though we have no guarantee that the level would be partitioned in such a way. It does not matter for this example because we know that each leaf in our BSP leaf tree will represent a convex region and, as such, the potential visibility set is actually calculating the visibility of each leaf. However, for the sake of demonstrating the basic premise, pretending that each room is a leaf makes it easier to identify the problem our PVS calculator will have to solve. Note that it is also assumed that a ceiling would exist over this level, but has been removed so that we can see inside.

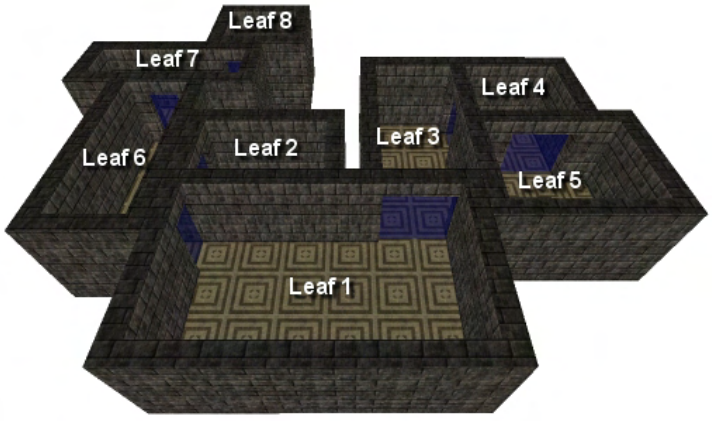


Figure 17.1

When this BSP tree is passed to the PVS calculator, the calculator has the task of determining every leaf that can potentially be seen from every other leaf. As this level consists of eight leaves, this means that eight visibility sets will be generated (one per leaf). Furthermore, the size of each of these arrays would also be 8 so that there is a byte element for each leaf. The eight bytes in leaf 1's visibility set would represent which of the eight leaves are potentially visible and should be rendered when the camera is located in leaf 1. Forgetting about the fact that arrays are zero indexed, byte 1 would always be set to 1 in the first leaf's visibility set since this is the byte which identified itself as being visible (i.e., every leaf will always be visible with respect to itself). Byte 2 would be set to 1 in leaf 1's visibility set if leaf 2 was potentially visible from leaf one. We can clearly see in Figure 17.1 however that this is not the case as there is no doorway (gap) leading from leaf 1 into leaf 2. As such, no line of sight between these two leaves is possible. This means that byte 1 in leaf 2's visibility set would also be set to 0 because, if leaf 1 cannot see leaf 2, leaf 2 cannot see leaf one either. We can see that byte 3 in leaf 1's visibility set would be set to 1 by the PVS calculator because there is an open doorway leading from leaf 1 directly into leaf 3. Consequently, leaf 3 would also have the first byte of its visibility array set to 1 as it can also see leaf 1. So, we have determined that after the PVS calculator has compiled the PVS, each leaf would have an array of leaf visibility information equal in size to the total number of leaves in the level. During the rendering of our level, any leaves that have their bytes set to 1 in the visibility set for the current leaf we are in should be rendered.

The reason that this data structure is called a 'potential' visibility set instead of an 'actual' or 'exact' visibility set is a result of the fact that it is calculated at development time and the resulting data is essentially static. In order to get the exact visibility set at the camera's current location in the game world, we would have to also factor in the camera orientation and the FOV. The PVS calculator is a tool that analyzes the geometry at development time and as such has no such concept as a camera (or similar runtime components). The camera is something that exists in the game world only after the game is running and is something that is dynamically changing as the player moves about the scene. As mentioned, the calculation of a PVS on a complex level can take many minutes or even hours, so we certainly cannot calculate it every time the player moves. Even if it were possible to calculate it in a respectable time, the whole idea of the PVS is that it removes any such runtime burden from the CPU. Therefore, the PVS for a given leaf will not describe what is currently visible from the player's location within the game world, but will instead describe everything that could possibly be visible from that given leaf.

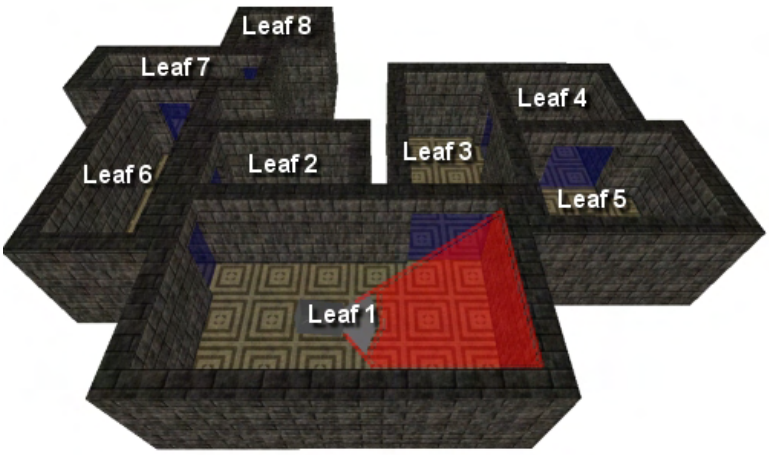


Figure 17.2

To understand the differences between the two situations, take a look at Figure 17.2 which shows the same level geometry. We also see where the camera might be located and oriented during a given frame update. We note that the camera is in leaf 1 and is oriented to the right. The only geometry that can actually be seen would be sections of the east and south wall of that leaf. The red tinted area describes the FOV given by the view frustum. As you can see, if we could pre-calculate this information, it would mean only a very small number of polygons would have to be drawn in a given update when compared to the total polygon count of the level as a whole. However, since this would involve factoring in the camera's position and orientation, which can change from frame to frame, this is something we cannot pre-calculate at compile time.

Although we cannot pre-calculate the exact visibility information for the camera at every given location in the scene using this type of PVS system, we can do the next best thing. For each leaf, we can calculate everything that can possibly been seen through all its exits. To do this, we will borrow ideas from the theory of light transport. That is, we will essentially fill the leaf with light and let it flood out through each of its exits. Any other leaves in the level that are touched by this light are added to the visibility set of the current leaf being processed. By filling leaf 1 with light for example and finding all the leaves in the level that get touched by that light, we essentially compute everything that could possibly be seen from any location within leaf 1. This completely covers the full range of orientations that the camera could be maneuvered to within that leaf to get the maximum view angle through one its doorways.

The process of shining this light out of the leaf though its exits is done for each exit in that leaf in turn. After we have done it for each of the exits in the leaf currently being processed, we will have successfully identified all the leaves that have the potential to be seen (i.e., the potential visibility set for

that leaf). This process is performed for each leaf in the level, resulting in potential visibility information being generated for each leaf. Collectively, we refer to this as the PVS for the game level.

To bring some clarity to the above explanation, take a look at Figure 17.3 which depicts a situation part way through generating the visibility set for leaf 1. The green tinted area filling leaf 3 and partially filling leaf 4 shows how the light would flow out of the north east exit of leaf 1 were we to fill it with light. This tells us that both leaves 3 and 4 can be potentially seen from leaf 1 were the camera to be looking through that exit during the game. The square inset in Figure 17.3 demonstrates this fact by showing a render of what the camera could actually see through that exit were it to be located directly in front of it. Therefore, we know that if the camera is ever found to be in leaf 1 during the game, leaves 3 and 4 would need to be rendered as they have the potential to be visible. Whether or not they are *actually* visible depends on where the camera is located in the leaf at run time and which direction it is facing. Were it to be in the same location as shown in Figure 17.3 but facing southwards, obviously neither of those leaves could be seen. However, calculating what has the potential to be seen in an occluded level narrows down the number of leaves that have to be processed. In the end, we will only have to work with a few of them in the typical case.

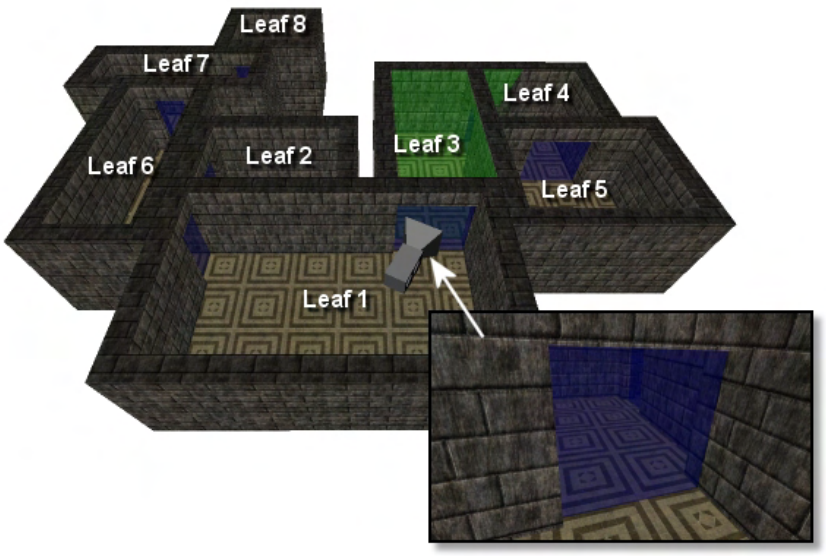


Figure 17.3

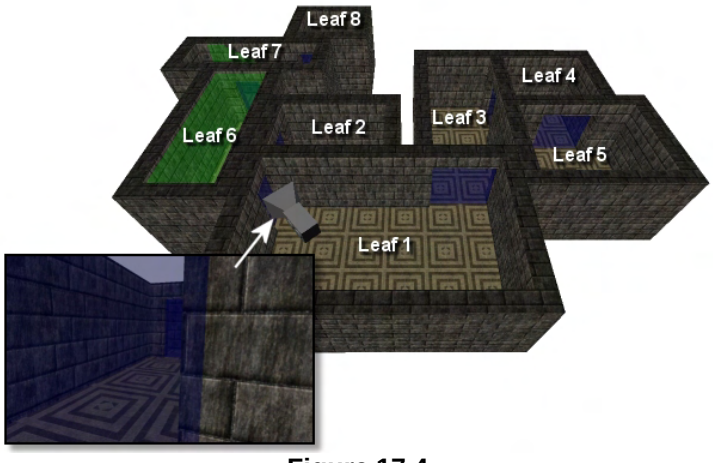


Figure 17.4

So, we can see above that after the PVS calculator has calculated the visibility for the first exit in leaf 1, it will set the bytes that correspond to leaves 3 and 4 in its visibility array to 1 (indicating potential visibility).

As leaf 1 actually has two exits, the PVS calculator would then have to perform the same step for the northwest exit. Once again, we model the phenomenon of light flooding out from the northwest exit and find that the light touches leaves 6 and 7. We can see that the light only partially touches leaf 7 through leaf 6's north exit, but that is all we need to know. Can any part of leaf 7 ever been seen when the camera is in leaf 1? In this case the answer is yes, and as such, when the camera is located in leaf 1, leaf 7 should also be rendered. The square inset in Figure 17.4 demonstrates that this is

the case by showing a render of what the camera might see if it was located in leaf 1 and looking through the northwest exit.

Figure 17.5 shows the visibility set that would be constructed by our PVS calculator for leaf 1. It contains itself (naturally) and the four leaves that light flooded into. In this small and simple example, this means that 5 of the 8 leaves would need to be rendered when the camera is located in leaf 1. While this does not seem like much to be excited about, we can easily imagine the savings that would be introduced when working with a proper game level of vast size and complexity. For example, imagine if leaf 8 had a doorway that led into a network of 100 other leaves, and leaf 5 also had an eastern exit that led to a network of an additional 250 leaves. Although the level would now be 358 leaves in size and would be much slower to render using brute force, with a PVS, only those same 5 leaves would ever need to be rendered as leaf 1 cannot see leaves 8 or 5 and thus, any of the leaves they might lead to.

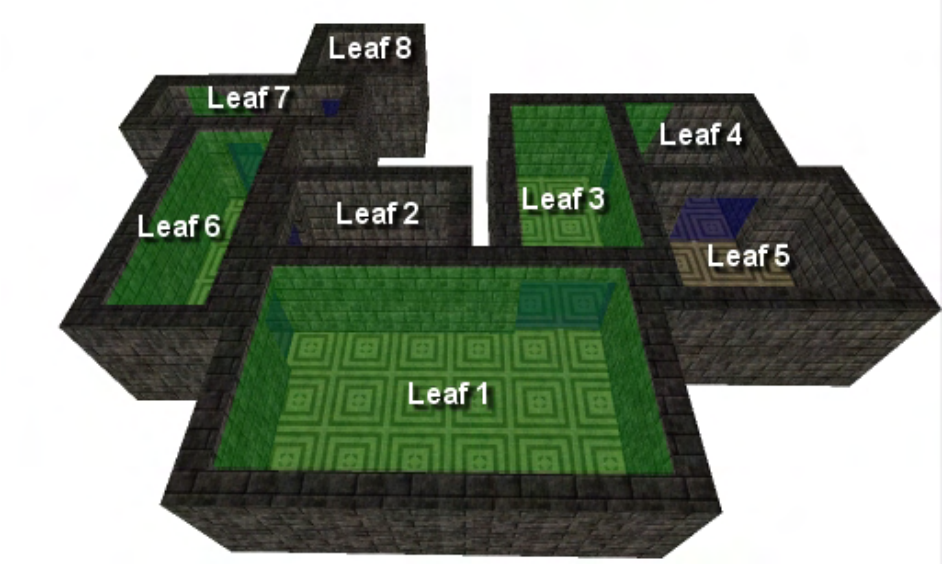


Figure 17.5

So the visibility set for a given leaf will provide us with all the leaves that could possibly be seen through the exits in that leaf. We can see in Figure 17.5 that this is always an over-approximation since we have to cover all bases and calculate everything that could be seen from every position/orientation that the camera might assume within that leaf. We can imagine that if the camera were located in the south region of leaf 1 in Figure 17.5 and was facing in a southerly direction, leaves 7, 6, 4, and 3 would not actually be visible since they would be behind the camera (and therefore, not contained in the frustum). Nevertheless, the visibility set for the leaf would still contain them.

> **Note:** Although we discussed the process of computing the visibility set as one of shining light out through the exits of each leaf, this was really just an analogy with respect to the process that will be performed. No lighting calculations of any kind are performed and no lights are ever placed into the scene during the process. We essentially simulate the same effect of light flooding using clip planes and portals as we will discuss later in the lesson.

Finally, before finishing up this initial review of how PVS data is calculated and what it represents, it is worth noting that it can be further refined using frustum culling at run time. This can often produce something that is much closer to an actual visibility set. In the previous example we said that if the camera was in leaf 1 and facing south, we would still have to render leaves 7, 6, 4, and 3 even though

they would be outside the frustum. Of course, once we have the leaves that are potentially visible from the current leaf, instead of just blindly rendering them, we can frustum cull their bounding boxes if we wish. In this case, 4 of the 5 leaves in leaf 1's visibility set would be culled away, leaving us to render only the one leaf. So, the PVS is a conservative pre-calculated visibility set for each leaf which can be fetched at run time for the current leaf the camera is in. After the visible leaves described by the PVS have been retrieved, we can further reject leaves from the visibility set by performing a frustum cull on those leaves. As we already know, frustum culling will factor in the camera's position and orientation, so we can reject any leaves in the PVS for the current leaf that are situated outside the frustum.

17.3 Rendering a BSP Leaf Tree with PVS

It might seem strange that we would discuss the technique for rendering a BSP leaf tree using a PVS when we have not yet discussed how to calculate that PVS data. However, the rendering process that uses the PVS data is completely separate from the development of the tool that is used to generate it (the PVS calculator). The rendering of the PVS equipped level data will be performed by the game itself and is actually extremely simple once we now how the data will be stored and loaded from file. Examining how our BSP leaf tree will draw itself using the PVS data will also help us understand the PVS calculator that we will need to write in order to build that data. Therefore, the topic discussed in this section is applicable to the application that ultimately ends up rendering the PVS data (the game's 3D rendering engine) and not the PVS calculator tool itself.

Rendering the PVS is delightfully simple. The general rendering algorithm works as follows on a per-frame basis:

- Traverse the BSP tree with the camera's position vector to determine which leaf the camera currently resides in. This is very fast because we are only be traversing a small fraction of the entire tree (thanks to the hierarchical nature of BSP trees). This phase returns 'Camera Leaf'; the leaf in which the camera currently resides.

- Each leaf has a PVS defined for it describing which leaves are visible. The size of this array for each leaf would be the total number of leaves in the tree. Loop through that camera leaf's PVS data. Any bytes in the array that are set to 1 indicate that we need to render the leaf (i.e. the polygons in that leaf). If Byte 23 is set to 1, then we render all of the polygons in Leaf 23 for example.

These two steps are all that is involved once the data has been stored in memory. Each leaf in the tree would have its own visibility array and therefore, all we have to do is find the leaf the camera is in and then iterate through that leaf's set rendering any visible leaves.

We will now look at some code demonstrating this process. Although the actual implementation will be discussed in the accompanying workbook, the code and pseudo-code in this textbook will stay close to the system we will be using. For example, in this next section of code it is assumed that the CBSPLeafTree class is derived from CBaseTree and as such, it is in the ProcessVisibility method of the derived class that the camera's leaf is located and its PVS iterated through. The function first issues a

call to CBaseTree::ProcessVisibility so that the leaf bins get emptied and then it issues a call to the CBSPLeafTree method called FindLeaf. This method is passed a (camera) position and returns the index of a leaf in the tree's leaf array. This index is then used to fetch a pointer to the leaf. This first section is shown below.

```
void CBSPLeafTree::ProcessVisibility (CCamera &Camera)
{
    CBSPLeaf *pCurrentLeaf=NULL, *pVisibleLeaf=NULL;
    UINT     CurrentLeafIndex;

    // Call base class to empty leaf bins
    CBaseTree::ProcessVisibility( Camera );

    // Get pointer to current leaf the camera is in
    CurrentLeafIndex = FindLeaf ( CCamera.Position );
    pCurrentLeaf = m_Leaves[CurrentLeafIndex];
```

Now that we have the leaf the camera is currently in we will walk through its PVS array. In this example each leaf is assumed to have a BYTE array called 'PVS' which has an entry for every leaf in the level which can be set to either 1 (visible) or 0 (invisible). In the final section of this function we loop through each element in this array and test to see if it is non-zero. If it is then it means that leaf is visible and should be rendered. The loop variable 'i' describes the current byte/leaf in the array we are testing visibility for. It is used to fetch a pointer to the leaf that needs to be rendered. We pass this leaf's bounding box into the CCamera::BoundsInFrustum method and if the leaf is inside the frustum we set its visible status to true.

```
    // Loop through PVS for this leaf
    for (int i = 0; i < m_nNumberOfLeaves; i++)
    {
        // Is leaf 'I' visible to potentially visible from current leaf?
        if (pLeaf->PVS[i])
        {
           //Fetch pointer to potentially visible leaf
           pVisibleLeaf = m_Leaves[i];

           // Is it actually visible (inside the frustum)
           if (CCamera.BoundsInFrustum(   pVisibleLeaf->BoundsMin,
                                          pVisibleLeaf->BoundsMax) ;
              pVisibleLeaf->SetVisible( true );
        }
    }
}
```

Remember that the CBaseLeaf::SetVisible call is the method that adds all the data in that leaf to the relevant leaf bins so that it is ready to be output to the frame buffer batched by subset when the application finally calls CBaseTree::DrawSubset. As you can see, the PVS idea fits in very nicely with our CBaseTree rendering system.

An important thing to remember is that the PVS array for each leaf is *view independent*. When we calculate the PVS it will contain all of the leaves that are visible from anywhere inside the current leaf. This does not mean that during our game the camera can necessarily see into all of those leaves at the

current moment in time. As mentioned, the nature of PVS it that it represents an *overestimate* of visible geometry and as such, frustum culling has been employed to further reduce the number of leaves that have to be rendered (to a mere few in the typical case of a highly occluded environment). This means that even if the game level contained 10,000 leaves, we would still only have to process and render a handful of them. As a result, the frame rate is essentially decoupled from tree size, as it was with the node compiler.

17.4 Zero Run Length Encoding

The problem with the code snippet discussed in the last section is scalability. If our game level had 5000 leaves then the total memory needed for just the PVS data alone would be:

5000x5000 bytes = 25,000,000 bytes = 25 MB (approx)

For most game programmers, that is simply too much memory to dedicate to this task, important though it may be. Remember that this does not account for the memory consumed by textures, polygons, or even the BSP tree itself. There is at least one fairly obvious thing we can do to reduce the memory requirements for PVS. For starters, because a leaf is in going to be in only one of two states (visible or invisible), we do not need a full byte per leaf to store that state. Instead we can store a single bit per leaf in the PVS. This reduces the PVS memory footprint by a factor of 8 (down to about 3 MB in this example) since each byte now stores visibility information for eight leaves instead of one. But even this is still a bit on the large side.

Imagine that you are located in a large installation of some kind (your game level) with the roof removed. Let us say that this installation has thousands of rooms, where each room is a leaf in our tree. In this case we know that most leaves will only be able to see a handful of other leaves (through doors, windows, etc.). We can deduce then that the PVS for any given leaf will contain mostly zeros, the exception being the leaves that are in close proximity which have direct line of sight with the source leaf.

Let us say that our example level has 1000 rooms and that we are in room (leaf) 0. Further, let us say that from leaf 0 we can only see into rooms 1, 2, 3, 20, and 35. Again, the PVS data for this leaf has to be large enough to hold one bit for every leaf in the tree. So our requirements will be (1000 + 7) >> 3 = 126 bytes. Do not worry if this number caught you off guard, we will explain how we arrived at this figure next.

We add 7 and then divide by 8 to deal with the truncation that happens when using a long integer. For example, if we had 9 leaves, we know that we need two bytes to hold this information. Byte 1 would hold the first 8 bits and byte 2 would hold bit 9. It is a common mistake to assume that to calculate the space needed we can simply do: 9 / 8 = 1. This would only give us 1 byte, which we know is not sufficient memory to store all of the information we need to in this example (9 bits). However, if we add 7 and then divide by 8 we get: 9 + 7 = 16 / 8 = 2 bytes. Thus we can always be sure that we round up to the nearest whole number and not down.

It is certainly wasteful to use 126 bytes for leaf 0's PVS when it can only see into five rooms, especially when we only need one bit per leaf. In this case, the visibility information for leaves 1, 2, and 3 would all be in byte 0 of the PVS array. The in-memory PVS data for leaf 0 (without Zero Run Length Encoding) would look like the scenario depicted below.

> **Note:** The array in this example is assumed to continue on to the 126th byte to represent the visibility bits for each of the 1000 leaves we are using in this example. Everything past byte 4 would be zero however, leaving 121 bytes of zeroes.

	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	.....
Value	15	0	16	0	8	0	0	0	0
Leafs	0-7	8-15	16-23	24-31	32-39	40-47	48-55	56-63	.....

Zero Run Length Encoding

If you are comfortable with binary numbers (and you should be) you should have no difficulty understanding the situation above. In this example leaf 0 can see leaves 0 (itself), 1, 2, and 3 and this information is all contained in the first byte (Byte[0]). We set bits 1, 2, 3 and 4 to '1', which produces the decimal value 15. We cannot see any of the leaves ranging from 8-15 so all of the bits in Byte [1] are left at zero giving us a decimal value of 0. The next byte in the array (Byte[2]) represents leaves 16-23 and we can see one of these leaves (leaf 20). Leaf 20's visibility bit is bit 5 in Byte[2] (16 = bit 1, 17 = bit 2, 18 = bit 3, etc.) so we set the fifth bit in Byte[2] to 1 and get a resulting byte value of 16. The next byte in the array represents leaves 24-31 and since none of these are visible from the current leaf, we leave the byte set at 0. Byte[4] represents leaves 32-39 and we can see leaf 35. So the appropriate bit in Byte[4] is set (bit 4) giving us a decimal value of 8. From this point on, all of the other bytes in the PVS data array would be zero because no other leaves are visible from leaf 0. For a very large level, this proves to be a tremendous waste of memory, especially when we consider that these wasted (zero) bytes occur on a per leaf basis. It also adds unnecessary overhead at runtime if the renderer is required to test all of these bytes when we know beforehand that the rest of the array is set to zero.

Now look at the list of numbers just below and you will see the same example represented using Zero Run Length Encoding (ZRLE). You will note that even our simple data set has been compressed from 126 bytes down to 9 bytes. The savings would be even greater on larger levels. This is the compressed format which our PVS calculator will use when generating the data and saving it to file. Our runtime component will therefore need to know how to interpret this data when calculating visibility. That is, the PVS data for each leaf will now be stored in a compressed format even in memory and we will have to learn how to decode it when we are testing a leaf's visibility array. Using this compressed format will help us keep the PVS memory footprint down to a minimum. The implications for runtime performance are clear as well since we will now need to examine only 9 bytes at every leaf instead of 120 bytes. See if you can spot the pattern in the following numbers:

15, 0, 1, 16, 0, 1, 8, 0, 121

When a byte does not equal zero, it is simply stored in the array as it normally would be (uncompressed). You can see that byte 0 which was equal to 15 does not get compressed and is instead just written to the data array. The bits set in this byte have the same meaning as before; if a bit is set to 1

then the corresponding leaf that maps to that bit is visible. However, when a 0 is encountered, instead of just writing it to the array we will count how many zeros follow it. In our example, there is only one zero at byte 1, so we write in the zero byte followed by an additional byte which describes how many zeros are in "the run". While this does not seem all that significant, you can see the savings more clearly at the end where we would have had a run of 121 additional zero bytes. This has been reduced to only two bytes (0 and 121). On very large levels, you are sure to find many such "runs" throughout the data set. Note that the maximum single run of zeros can be at most 255 because that is the largest value that can be stored in a single byte.

Although the last code snippet demonstrated that each leaf structure had its own visibility array, our PVS calculator will actually store all the visibility information for every leaf in a single PVS data array that is owned by the tree. This means the PVS data for the entire block would be stored in a single continuous array. That is, leaf 1's visibility bits will be followed by leaf 2's visibility bits which will be followed by leaf 3's visibility bits, and so on. The leaf structures will store indices into this master PVS array describing the byte index where that leaf's visibility bits begin.

With all of this in mind, we will look at a revision of the previous function which has been adapted to index into the master PVS data array to fetch the visibility information for a given leaf. Once the start of that information has been located, we will iterate through it in its compressed form and flag any visible leaves as being so.

The first part of the function is the same and calls the base class ProcessVisibility method to empty the leaf bins. The CBSPLeafTree::FindLeaf method is then used to fetch the index of the leaf the camera is currently in. This index is then used to fetch a pointer to that leaf.

```
void CBSPLeafTree::ProcessVisibility( CCamera &Camera)
{
     CBSPLeaf *pCurrentLeaf=NULL, *pVisibleLeaf=NULL;
     UINT     CurrentLeafIndex;
     int i;

    // Call base class to empty leaf bins
    CBaseTree::ProcessVisibility( Camera );

    // Get pointer to current leaf the camera is in
    CurrentLeafIndex = FindLeaf ( CCamera.Position );
    pCurrentLeaf = m_Leaves[CurrentLeafIndex];
```

Next we get a pointer to the start of the tree's master PVS data array which, in this example, is assumed to be stored in a byte member array called m_nPVSData. The current leaf will store a PVS index member (which will be calculated by the PVS calculator tool) which will describe the location from the start of the master PVS array where the current leaf's visibility bytes begin in this array. This offset is fetched from the leaf and used to increment the PVS pointer so that the local variable PVSPointer now points to the start of the PVS data for the current leaf.

```
    // Get pointer to tree's master PVS array (compressed byte array)
    BYTE *PVSPointer = m_nPVSData;
```

```
    // Find where this leaf's PVS begins in master set
    long PVSOFFSET   = pCurrentLeaf.PVSIndex;
    PVSPointer += PVSOFFSET;
```

We can no longer just set up a loop to iterate through the number of leaves since there will no longer be a byte in this array for each leaf in the tree. In fact, because the visibility information for the current leaf has been zero run length encoded, we really do not know beforehand how many bytes this leaf's visibility information will span in the master array. So we will have to keep track of the current leaf we are testing in the local variable nCurrentLeaf. This will start off at zero and we will break from the loop when we have processed all the leaves in the current leaf's visibility set.

```
    // Search through all leaves starting at zero
    long nCurrentlLeaf = 0;

    while (nCurrentLeaf < m_nNumberOfLeaves)
    {
```

We will now test the current byte being pointed to by PVSPointer, which will be the first byte in the current leaf's visibility information in the master PVS array in the first iteration of the loop. If this byte is non-zero then it means that this byte does describe some visible leaves (i.e., it has some of its bits set to 1) and therefore is not a compressed zero length run. When this is the case, we set up a loop to iterate through the 8 bits.

```
      if (*PVSPointer != 0)
      {
          for (i=0;i<8;i++)
          {
              BYTE mask = 1 << i;
              BYTE pvs = *PVSPointer;
              if (pvs & mask)
              {
                  //Fetch pointer to potentially visible leaf
                  pVisibleLeaf = m_Leaves[nCurrentLeaf];

                  // Is it actually visible (inside the frustum)
                  if (CCamera.BoundsInFrustum(  pVisibleLeaf->BoundsMin,
                                                pVisibleLeaf->BoundsMax ) ;
                   pVisibleLeaf->SetVisible( true );

              }
              nCurrentLeaf++;
          }// end for i;

          // Advance pointer to next byte
          PVSPointer++;
       }
```

As the above section of code shows, in each iteration of the inner loop we build a byte mask. This is a mask that is all zeroes except for the bit we wish to test. When a bitwise AND with the current PVS byte we are testing has a non-zero result, it means this bit must be set to 1 in the PVS byte and therefore the leaf it corresponds to (nCurrentLeaf) might be visible. Thus it should be frustum tested and, depending

on the outcome, flagged as visible. As you can see, just inside the bottom of the inner loop we increment the nCurrentLeaf counter variable as we step through each bit in the byte. This is our way of keeping track of how many leaves we have processed and therefore, which leaf the current bit we are testing corresponds to.

Now we see the final section of the function which is executed when the current PVS byte is found to be zero. This means that this byte is the start of a 'zero run' and the next byte that follows it will tell us how many zero bytes were collapsed into this single byte. This is very important for us to know because if the byte following the zero is 10 for example, this means that the zero byte actually represents a zero for 10*8=80 leaves. As such, we should increment our nCurrentLeaf variable by this amount.

```
       else
       {  // we have hit a zero so read in the next byte
          // and see how long the run of zeros is
          PVSPointer++;
          BYTE RunLength = *PVSPointer;
          PVSPointer++;
          CurrentLeaf += RunLength * 8;
       }

     } // End while
}
```

As you can see, because we only enter this code block when the current PVSPointer points to a byte that is zero, we know that the next byte following it will be the run length. So we increment the pointer and fetch the run length value. As this represents a compressed run of zero bytes and each byte holds the visibility information for 8 leaves (8 bits) this means we have just avoided testing a potentially large number of leaves. For example, if the run length following the zero byte was 20, this means we have just avoided having to test 20*8=160 leaves and we have also avoided 160 loop iterations. Of course, we must increment our nCurrentLeaf counter by this amount so that we know how many leaves we have skipped and what leaf the first bit in the next byte maps to.

> **Note:** The above code is for demonstration purposes only and will differ from the actual code explained in the accompanying workbook. All code in this textbook should be considered only placeholder and is shown to clarify the processes being discussed.

You should take away four things from the code snippet above.

- Each leaf has a PVSIndex variable that describes how far into the master PVSData array its own PVS begins. Remember that each leaf has its own PVS, but that all of the leaves will store their PVS data together in one master array. It is this master array that is commonly known as the PVS and it describes the visibility of all leaves from any other leaf.

- If a non-zero byte is encountered, then we loop through all 8 bits in that byte, rendering the polygons in any leaf which has its bit set to 1. Remember that each bit in that byte represents a leaf, not a single polygon.

- If a zero byte is encountered in the PVSData array, we adjust the CurrentLeaf index variable to compensate for how many leaves *would have been* represented by the bits in the zero run. This shuttles us past all the zeros and speeds up the main rendering loop.

- The rendering loop exits once nCurrentLeaf > m_nNumberOfLeafs. This means we have processed the visibility bits for every leaf in the current leaf's visibility set.

We now have a good understanding of how the PVS data will be used by the runtime component to efficiently process only potentially visible geometry. Now might be a good time to run Lab Project 17.1 to see PVS in action. If you study the accompanying source code and workbook for this lab project you will also see that the additions to the leaf tree are extremely light. We have added a new member to our leaf structures so that they can now store an index into the master PVS data array describing where each leaf's visibility information begins, and we have also added code to read in the PVS data from file. Of course, this file is created by a development tool which we started to implement in the previous lesson. In this lesson we will now add a PVS calculator to our BSP leaf compiler. This will be the component of the compiler that will generate the PVS data for the compiled tree and save the zero run length master PVS data array out to file.

From this point on, all the processes discussed in this textbook will pertain to the compilation process of the PVS data in the compiler tool. That is, everything discussed from this point on will not be done at runtime, but will be part of our development cycle. The single goal of the PVS compiler is to create the PVS data array. While this is seemingly simple in theory, writing a PVS calculator is actually a non-trivial job. So consider yourself warned that creating a PVS is not an easy undertaking from the coding perspective. However, it is one of those tasks which is deceptively hard when you are struggling to understand it and then later seems so obvious and simple once you do. The rest of this chapter will make that clear.

17.5 The PVS Calculator

The process of calculating a PVS for BSP trees can be essentially broken down into two steps. These steps are performed after the leaf tree has been compiled. That is, our compiler will first build the BSP leaf tree and, instead of saving the data straight out to file as was the case in our previous lab project, the following two processes will now be performed.

- **Portal Generation**

- **Anti-Penumbra Clipping**

Since you may not be familiar with either of these terms, we will start with step one, 'Portal Generation'. In the coming discussions we will begin to see how these two concepts work together to create the PVS we are aiming for.

17.5.1 Portals Overview

In order to determine which leaves in the tree can see each other, we need to know about the gaps (exits/spaces) that exist between those leaves. For example, if we have two rooms with a doorway between them, we need to know how much of Room 2 can be seen through the doorway while standing in Room 1. The problem is that this is not information that we have automatic access to in our 3D world dataset. In fact, our polygon set tells us the opposite – where the gaps are *not*, instead of where they actually are.

To solve this problem, we are going to create temporary polygons called **portals** that are going to fill these gaps (they will be the same size and dimensions of all the gaps) between all of the leaves in the tree. In the two rooms with a doorway example, we somehow need to create a polygon that would fit into the doorway -- to "plug it up" so to speak. These portals will not be rendered or optionally not even kept around after the PVS has been calculated. The important role they serve is that by getting the dimensions of the portal which plugs up the doorway, we wind up getting the dimensions of the doorway itself, which is exactly what we need.

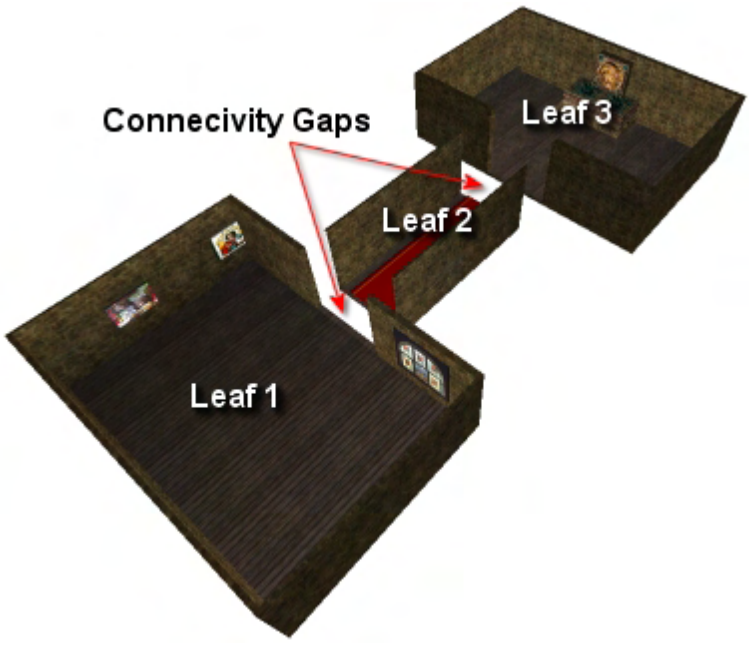


Figure 17.6

Although the BSP tree compiles the level into convex areas, the empty space regions are not typically closed hulls as Figure 17.6 demonstrates. We can see for example that leaf 1 represents a convex area of empty space and the polygons contained in leaf 1 lay on the bounds of that convex region. However, we can also see (note that we have deliberately pulled the construct apart into 3 distinct pieces for clarity) that there is a hole in the polygon data surrounding leaf 1's empty space. This is the hole/gap that leads into leaf 2 and thus describes the possible visibility from leaf 1. What we need to do is find a way to build temporary polygons to plug these gaps (a little like using them as doors). Of course, while this example assumes that each leaf is a separate room, this need not be the case depending on how the tree was built. All we know is that a leaf is a convex region of empty space that is partially bounded by polygon data. Anywhere there is a gap in this bounding data, the leaf has visibility out into a neighbouring leaf and it is this information we need to know. Thus we need to build portals for every single gap between all the leaves so that we have exact information about which leaves connect to which other leaves.

Once these portals are created, we then enter the second phase: anti-penumbra clipping. Without going into detail at this present time, let us just say for now that the edges of these portals are going to be used to build frustum-like shapes so we can determine exactly what a leaf can see through each of its portals. This is a somewhat fast and loose description, but will suffice at this time.

As you can probably imagine, creating portal polygons to fit into all of the gaps in our level results in a very high portal count. If a room has 5 doorways, we will create a portal in each doorway. This is going to be true for every leaf, so we expect to see many portals created along the way.

One might also imagine that we can create these portals given the information at our disposal in the BSP tree. In fact, the BSP tree tells us everything we need to know. In the previous lesson we discussed how the split planes of the BSP tree will, by their very nature, divide a non-convex region into a series of convex ones. This is because the planes are created from the actual polygon data and thus any non-convex object will have one or more polygons whose planes cut through the geometry.

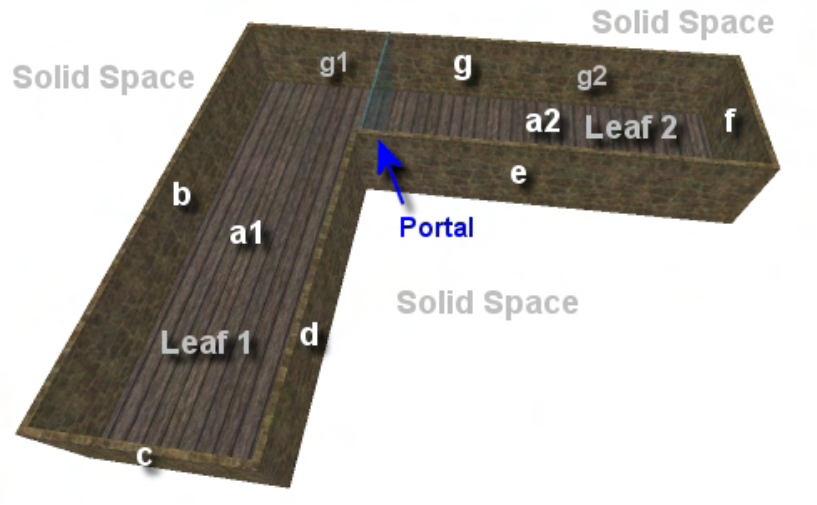
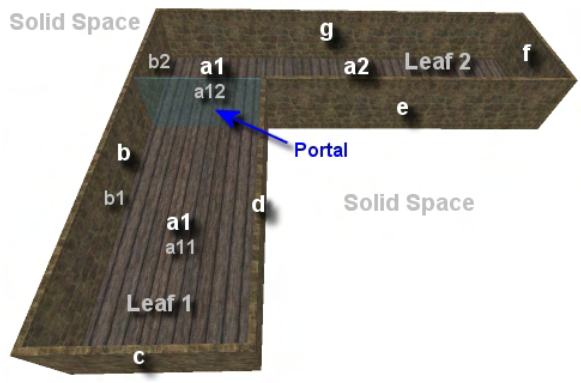


Figure 17.7

To demonstrate this fact, Figure 17.7 shows a simple non-convex mesh (roof removed). The labels given to each wall identify the polygons and the order in which they were chosen (alphabetical) as node planes during the BSP compiler. For example, polygons a1 and a2 represent two polygons that lay on the same plane -- plane A, used as the root node of the tree. Assuming that the split planes were chosen in alphabetical order, we can see that only when we get around to choosing polygon d as the split plane does the geometry get subdivided by this plane. We see that this cuts the polygon g into two g1 and g2 and splits the non-convex object into two convex objects. Because this split is caused by node d in this example, which created a leaf on either side, we know that a portal will definitely exist on this plane (in this example). We do not know how big this portal should be yet and we will get to that in a moment. But we can see that since node d was the node that split the geometry into a leaf on either side, this node must be where the portal exists between those two leaves. The portal we would ultimately need to generate in this example is shown in the diagram. Of course, the exact same construct could be compiled using a different plane selection order which would generate a completely different tree. That is, the splits would happen at different planes and the leaves formed would be of different dimensions to those shown in figure 17.7. This also means the portal that bridges those leaves would be in a different location.

> **Note:** Although all of the following diagrams are shown with back face culling disabled, all polygon normals are assumed to be facing inwards towards the center of the leaves in which they belong.

As one example, imagine that the above geometry was compiled into a tree such that polygon e was selected prior to polygon d. This means the split into two convex leaves would happen along plane e instead of along d. Consequently, the portal would be along this plane instead, as shown in the top-most image in Figure 17.8. In fact, to make sure we definitely understand this relationship, let us build the BSP tree in that section so that polygon e is selected prior to d. Although we have discussed building solid leaf trees in the previous lesson, we will still show step by step how this particular tree is constructed so that we can analyze the tree properties and recognize where portals can exist.



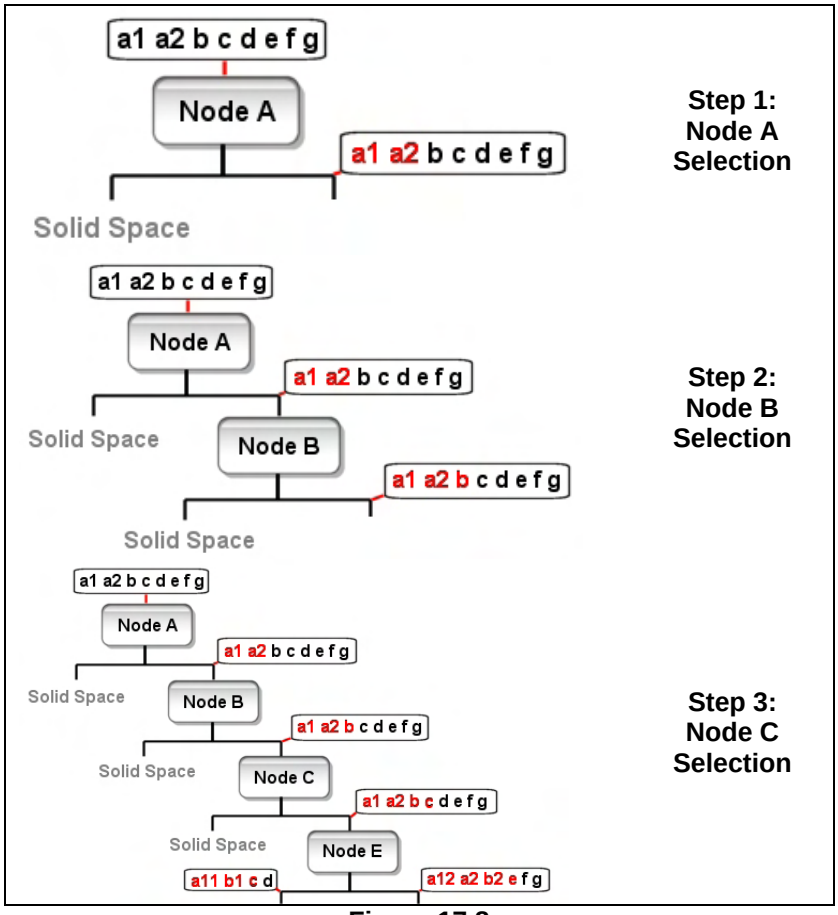


Figure 17.8

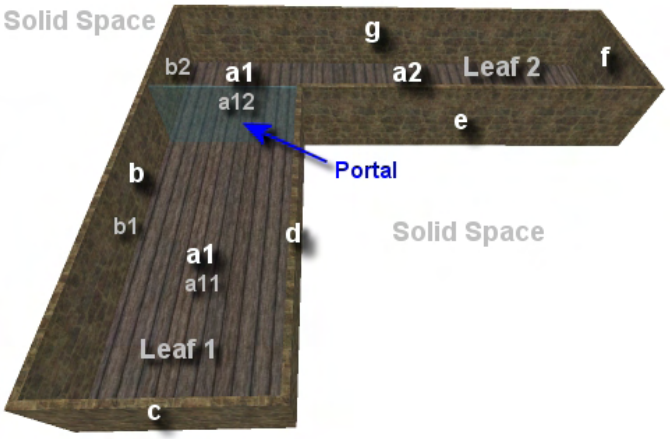
The BSP tree compiler is passed a list of polygons (a1,a2,b,c,d,e,f,g). At the moment, this is a single non-convex mesh. At the root node in this example polygon a1 is selected and its plane (labeled A) is stored in the root. Polygon a1 is then marked as a splitter so that it is not selected as a split plane again further down the tree. It is then added to the front list. The rest of the polygons in the list are also classified against plane A and are all determined to belong in the front list. We can see that in the diagram all the polygon data is either on or above the floor plane. During the classification test polygon a2 is found to be on the same plane as A, so it too is marked as having been used as a splitter so that it is not selected again. We can see in step one that all the polygons are passed into the front list and polygons a1 and a2 are highlighted red so that they are not used as a split plane again. Because no polygons exist in node A's back list, its back pointer is set to null and we know that this means it is solid space. This makes sense when looking at the diagram as this would represent the region of space underneath the floor.

In step 2, polygon b is chosen as the splitter and is marked as such before being added to the front list. All the polygons in the list are either on plane (with matching normals) or in front of plane B, so they all get added to the front list. The space behind polygon b (node B) is identified as being solid space.

In step 3, polygon c is chosen and the same process is repeated; all of the polygons are added to the front list and the halfspace behind node C is identified as being solid. The front list is passed down to the front child where, in this example, polygon e is chosen as the next splitter, creating node E. If you refer back to the geometry diagram, you will see that this node carves its input polygon set into a front and back lists, splitting polygon b into two child polygons labeled b1 and b2 and splitting polygon a1 into

polygons a11 and a12. Polygons a11, b1, c, and d are assigned to the back list as they are clearly situated behind polygon e and polygons a12, a2, b2, e, f, and g are passed into the front list.

For the sake of simplicity let us process the back list first. As we step into the back child we find that only polygon d has not yet been selected as a splitter, so it is selected, creating node D. At this point, polygons a11, b1, c, and d have all been used as splitters. We classify them against node D and find that they all belong in the front list. This identifies the back of node D as being solid space. As there are no polygons in the front list which have not been used as splitters, this means we have reached a terminal node and a leaf is created. In the diagram, this is referred to as leaf 1 and as we can see, it contains the polygons that made it into that leaf (a11, b1, c, and d).



At this point we return from node D (having created a leaf there) and find ourselves back up at node E. Node E's back tree has been built, but we have not yet recurred into its front child. In the diagram we can see that polygons a12, a2, b2, e, f, and g are passed into the front child with only polygons f and g remaining as potential splitters. In this example polygon f is chosen as the splitter and the polygon list is classified against the node. All polygons are found to exist in the front space which creates an empty back list, thus identify the backspace of node F as being solid space.

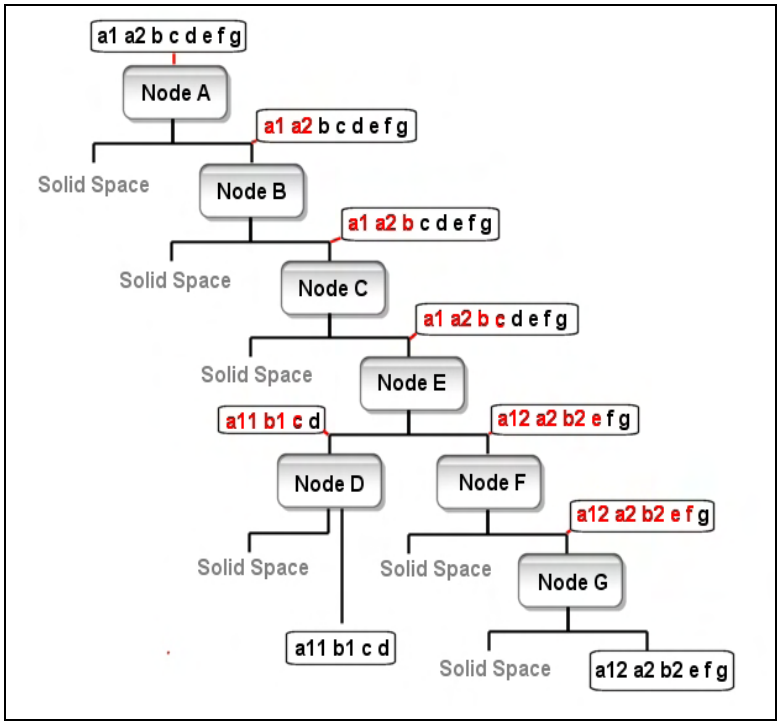


Figure 17.9

The front list is passed down into a final child where the only polygon not yet selected as a splitter (polygon g) is selected, creating terminal node G. Classifying the input list of polygons against node G reveals an empty back list and thus solid space exists at node G's back halfspace. All the polygons end up in the front list, but because they have all been chosen as splitters, this must be a terminal node and a leaf is created here and the polygons are added to it. The final tree is shown in Figure 17.9.

When analyzing this tree, something starts to become very clear. A portal is a polygon that bridges the gaps between two leaves and therefore, is a polygon that, if fed into the tree, should end up existing in both of those leaves. Therefore, the only nodes in the tree that could possibly create valid portals are the

nodes which have front *and* back children. A node that has only one child could not possibly create a valid portal because a portal by its very nature must have a leaf both in front and behind it. These are the leaves for which the portal describes connectivity. Therefore, we can see by looking at our tree that the only node plane on which a portal could possibly exist in this example is node plane E. All other nodes have solid space behind them. Thus, any portal created on that node plane would have a leaf on one side and solid space on the other.

Since a portal must have a leaf (empty space) on either side, our first step in the portal generation process will be to identify which nodes in the tree have both front and back children. These are the only nodes where portals can exist and are the only nodes that we will consider in the portal generation process. This typically rejects somewhere in the region of about 50% of nodes having to pass through the portal generation process. Imagine for example that we created a polygon that lay on the plane of node E and was the size of the portal depicted in the diagram. If we were to pass this polygon through the tree (performing portal/plane classifications at each node) the polygon would make its way into node E. At node E it will obviously be found to be on plane and could be sent down both the front and back trees of that node. This portal polygon would eventually end up popping out in both of the leaves on either side of node E and thus is a valid portal (i.e., a portal that exists in two leaves).

It is important to note that this does not mean that every node plane that splits the geometry into two lists will always create a valid portal. For example, imagine compiling the geometry shown in Figure 17.10 into a BSP leaf tree. In this example we can see that at some point during the BSP compilation process the long wall polygon of the closest inward facing cube would be selected as a splitter. This plane clearly has polygons both behind it and in front and as such, would be a node that has two children. However, we can also see that no portal should be created on this wall as it clearly does not connect the two leaves (cubes). Admittedly, this is an extreme case since in this particular scenario there is no way for the player to navigate from one region into the other, but it does highlight the fact that not every plane that has front and back children will create a valid portal. However, we do know now that nodes that have only a single child cannot possibly create portals. Therefore, we should not even try to do so for these nodes. Only the nodes in the tree that have two children are candidates for portal generation, although we now recognize that only some of them will actually create valid portals.

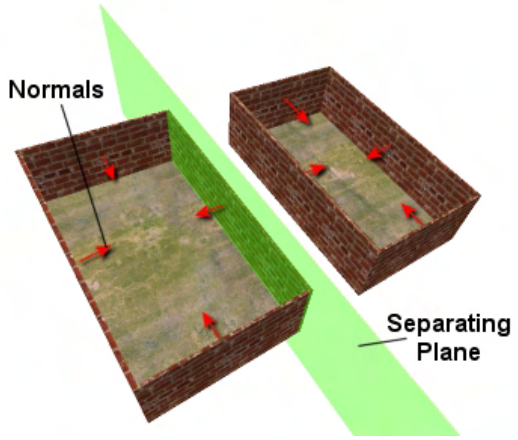


Figure 17.10

Although we still have no idea how to create a portal at this point, we have discovered some very important facts necessary to understanding the portal generation process:

- A portal is a polygon that ‘plugs’ the gap between two leaves.
- If a valid portal polygon was fed into the tree and sent down both sides of a node when it was determined to be on plane with that node, the portal would exist in two leaves.
- All portals throughout the entire level will exist on one of the tree’s node planes.
- Portals will never exist on the planes of nodes that have only a single child.

- We only need to create portals at nodes that have two children, although not all portals generated will necessarily be valid.

17.5.2 Portal Generation Overview

In this section we will discuss the steps involved in the generation of portals. Since this can seem to be a very complex topic at first, we will use a lot of diagrams to help you visualize the process.

At a high level, creating a complete set of portals for our Solid Leaf BSP Tree involves the following steps:

1. Visit each node in tree (node[i])

2. If node[i] does not have front and back children, skip this node

3. Generate a huge polygon (i.e., portal) on the plane of node[i]. This polygon should be large enough so that it encompasses all of the polygons in the level along that plane. How large it is does not matter just as long as it is large enough so that it spans the entire level along that node plane. This polygon represents an initial portal that we will later send through the tree and clip down to the appropriate size. A portal structure will be used to contain the information for a portal. This will be identical to a polygon structure except that it will have a few extra members so that it can store the two indices of the leaves it ends up in (if does turn out to be a valid portal). It will also store a 'next' pointer so that multiple portals fragments can be linked together and returned from the portal clipping function.

4. Once we have created the initial large portal for node[i], we send it down the BSP tree starting at the root node. Because a solid leaf tree is divided into solid/empty areas, clipping the portal against other nodes is easy. Any portal fragments that end up in solid space can be removed (because a portal cannot exist in the middle of a wall). As the portal gets recursively split, any fragments that end up in solid space are deleted and we are left with the shape and size of the portal as it should be to fit the gap between the two leaves the portal is being generated for. The actual clipping of the portal to the tree can seem a little complex. We will discuss it in some detail here and in greater detail later when we examine some code. Clipping involves traversing each node in the tree and testing the portal against the node plane to see whether it is to the front, back, spanning, or on the plane. Because the initial portal may get split into many valid portals as it is clipped to the tree, the portal clipping function will accept an initial portal as input and will return (potentially) a linked list of valid fragments that survived. Each fragment is a valid portal that ended up in two leaves. Of course, the function may return null if no portion of the initial portal survived. This means that there exist no valid portals on node[i].

   The portal clipping function will make extensive use of many of many of the skills we have already learned (e.g., polygon/plane classification and polygon splitting). The portal will be passed into the root node and will be classified against each node plane as it passed down the tree. Again, the portal can be either in front, behind, spanning, or on plane with the current node being visited. Here is an overview of what we do in each case:

- **In Front**

  If the portal passed into the current node being visited is found to be contained in the node's frontspace, the following two conditionals are executed:

  - If this is a terminal node then it means that the portal has ended up in empty space (a leaf). The index of the leaf in which it has landed is recorded in the portal's leaf index array. There will only ever be two elements in this array as a portal can never exist in more than two leaves.

  - If the front child of the current node is not a leaf, then the portal is passed into the front child recursively. As the clipper function will return a portal pointer (or a linked list of portals), the function then returns the portal pointer returned from its front child back to its parent. The portal that was passed into the front child may have been recursively split and clipped from the front branch of the current node's tree, so the pointer it gets back from its children may be a list of surviving fragments from the original portal that the current node passed into the child.

- **Behind**

  If the portal is found to exist in the backspace of the current node being visited, the following two conditionals are performed.

  - If no back child exists, then it means that the portal that has been passed into this node by the parent (or the application in the case of the root node) has ended up in the solid space behind the current node. The portal is deleted from memory and NULL is returned from the parent. This result is propagated up the tree through its chain of parents.

  - If the back child of the current node exists, then the portal is not in solid space and is passed into the back child node recursively. Any of the fragments that survive the portal being clipped to the back tree of the current node will be returned via a linked list or portal fragments which is then passed back to the parent.

- **Spanning**

  This is an interesting case as it is only this case which is responsible for the initial portal being split into multiple fragments. As those fragments are then passed down the front and back trees respectively, where they themselves may get split when found to be spanning other planes in the tree, we have a cascade effect where a single initial portal may get split into dozens of valid portals during the recursive clipping process.

  When the portal is spanning the node, the portal is split by the node plane. This returns two new child split portals and the original portal is deleted. What is ***very***

important however is that the child portals inherit the leaf array from their parent. That is, if the parent being split was found to have existed in leaf 5, both child portal fragments will also have this leaf index stored in them as well. This way, should the child portals end up in leaves of their own, we know the two leaves each fragment ended up in. What happens to both the front and back split portals is discussed below.

- The Front Split
    - If the current node is a terminal node then it means the front split has made it into a leaf. The leaf index is determined and stored in the portal. This may be a valid portal if it ends up in another leaf later in the process.
    - If the current node has a front child then the front split portal is passed into the front child and the portal fragments that survive that branch of the tree are returned (in a linked list).
- The Back Split
    - If the current node has no back child then it means solid space exists behind this node and therefore the back split portal ended up in that solid space. When this is the case, the back fragment is deleted.
    - If the current node has a front child then the back split portal is passed into the front child and the portal fragments that survive that branch of the tree are returned (in a linked list).

After the front and back splits have been processed, we may have two linked lists of fragments (returned from the front and back sub-trees at the node). When this is the case, the two lists are stitched together and returned to the parent. If only a front list or a back list was returned, then just that single list is returned.

- **On Plane**

The on plane case is perhaps the most complex to deal with. Essentially we have to send the portal down both the front and back children (if they exist). However, the portal must be passed down the front first, which will return a list of any portal fragments that survived the front of the tree. If any portals fragments were returned from the front traversal, then each one in turn is sent down the back of the tree. What we are essentially saying in such a scenario is “first find out if any part of this portal ends up in a leaf in the front tree, and if so, find out if any of those fragments also end up in leaves in the back tree”. If so, then this plane is the plane that created a split between leaves and these portal fragments exist in those leaves. Of course, the node plane being tested may have solid space behind it, or it may have a leaf in front of it, so the basic process performed in the on plane case is as follows:

- If the node has a front child then the portal is passed into the front child and we get back a list of portal fragments that survived the front of the tree.

- If the node has no front child then the portal has ended up in a leaf and the leaf index is stored in that portal.

- If no portal fragments survived the front tree (and if the portal did not end up in a leaf at this node) we return NULL from the function as nothing survived down the front tree (i.e., it ended up in solid space).

- If there is a list of portals that survived the front tree, then loop through each portal in the list and pass it down the back tree. The portal fragments returned (if any) from each fragment being passed down the back are all collected such that, we have a list of all fragments that survived both the front and back sub-trees at the current node.

5. When the portal clipper function has returned all surviving portal fragments for node[i] (the current node having a portal generated for it), any portal fragments that survived will contain the index of two leaves in which that fragment belongs. Each portal in the list is then added to the tree's main portal array since they are valid portals.

6. Finally we return to step one and do the same thing for each node that has two children. At the end of this loop we will have created all the portals that plug the gaps in the entire level between leaves and our portal generation process will be complete. We will now have the means with which to calculate the PVS.

The above list of steps provided only a high level view of what has to be done, but do not worry, since we will not leave it at that. Next we will step through some examples of the portal generation process with diagrams to clarify the process. We will follow a portal as it makes its way through the portal clipping process and see exactly why the rules we have set out above generate the correct results. Before getting deep into the processes involved, let us have a look what the top level portal generation process might look like with some placeholder code. The function that we have called ProcessPortals in this next example is passed a CBSPLeafTree pointer to a compiled tree that contains the level data we wish to generate portals for. A PVS compiler tool would issue a call to this function after the leaf tree has been compiled.

In the following code CPortal is a specialized polygon class that has the following members, in addition to an array of vertices and a face normal.

```
CPortal        *NextPortal;
unsigned char   LeafCount;
unsigned long   OwnerNode;
unsigned long   LeafOwner[2];
```

Next Portal
This is a mechanism which allows us to stitch multiple CPortal structures together in a linked list during the recursive clipping process.

LeafCount
This will contain the number of leaves the portal was found to exist in. For a valid portal, this will be exactly 2. Any portal that does not end up in two leaves will be removed.

OwnerNode
This contains the index of the node that was used to create this portal. We will see why this is necessary during the clipping process. What is meant by OwnerNode is the node that the initial portal was created on prior to being clipped to the tree.

LeafOwner[2]
After the compile process, this will contain the indices of the two leaves whose gap is plugged by the portal.

> **Note:** It is assumed throughout the code snippets in this chapter that the nodes of the tree are also stored in a linear array along with the node planes and leaves. This is similar to how we implemented our BSP leaf tree compiler in the previous lab projects.

Let us now look at what a top level function in a portal processor might look like.

```
HRESULT ProcessPortals( CBSPLeafTree * pTree )
{
    HRESULT      ErrCode;
    CBounds      PortalBounds;
    CPortal    * InitialPortal = NULL;
    CBSPNode   * CurrentNode   = NULL;
    CBSPNode   * RootNode      = NULL;
    CPlane     * NodePlane     = NULL;
    CPortal    * PortalList    = NULL;


        // Store required values ready for use.
        RootNode = m_pTree->GetNode(0);
```

As can be seen above, the first thing we do is get a pointer to the root node of the tree, which will always be stored in element zero of the tree's node array.

In the next section we see the start of the main loop that will iterate through all the nodes in the tree's node array and generate initial portals on the node planes that have both front and back children. Inside the loop you can see that we use the loop index to fetch the current node being examined and then use the node's plane index to fetch the node plane itself. If the current node does not have a back child (or leaf) then it means that this node cannot possibly be a portal generator because it does not have leaves in both its halfspaces. You will recall from the previous lesson's workbook that a back node that has solid space behind it is assigned the value BSP_SOLID_LEAF, which is defined as the smallest negative floating point number we can assign.

> **Note:** BSP_SOLID_LEAF is defined as hex value 0x80000000. We make this the smallest possible negative number so that it does not conflict with other negative numbers we store in a node's back child index. Remember, this represents the index for a leaf in the tree's leaf array that exists behind that node.

```
        // Create a portal for each node
        for (unsigned long i = 0; i < pTree->GetNodeCount(); i++)
        {
            // Store required values ready for use.
            CurrentNode = m_pTree->GetNode(i);
            NodePlane   = m_pTree->GetPlane(CurrentNode->Plane);

            // Skip any that have solid space behind them
            if ( CurrentNode->Back == BSP_SOLID_LEAF ) continue;
```

In the next section we are now going to create a new portal on the current node plane that is large enough to fill (on that plane) the bounding box of the root node (the entire scene). As you can see in the following code, we fetch the bounding box of the root node and pass it (along with the node plane) into the portal's GenerateFromPlane method, which we will discuss in a moment. This will build the vertices for the portal such that the initial portal is a single polygon that fills the entire root node.

```
            // Allocate a new initial portal for clipping
            InitialPortal = new CPortal;

            // Should we build a full portal or not
            PortalBounds = RootNode->Bounds;

            // Generate the portal polygon for the current node
            InitialPortal->GenerateFromPlane( *NodePlane, PortalBounds );
            InitialPortal->OwnerNode = i;
```

When the function returns, the portal will be ready for clipping to the tree. Before we do that however, the portal is assigned the index of the node whose plane was used to create it. We will see later that this will be useful information to have during the clipping process.

At this point we are ready to send the portal into the tree at the root node. In this code we pass the initial portal and the root node index (node 0) into the ClipPortal method.

```
            // Clip the portal and obtain a list of all fragments
            PortalList = ClipPortal(0, InitialPortal );
```

This method is assumed to clip the initial portal to the entire tree and return a list of portal fragments that survived the process and ended up in two leaves (PortalList). At this point we can set the InitialPortal local pointer to NULL as we have another pointer to the list of fragments. We can now add these fragments to the master list of portals being compiled. In this example we do this using the AddPortals function. This function is assumed to take a linked list of portals and add each one to the master portal list.

```
            // Clear the initial portal value, we no longer own this
            InitialPortal = NULL;

            // Add any valid fragments to the final portal list
            if (PortalList)
            {
                AddPortals( PortalList );
```

```
            } // End If PortalList

          } // Next Node
    return BC_OK;
}
```

And that is essentially the top level process. As you can see, this is done for every node that has more than one child. By the time this function returns, a master list of portals will have been compiled and is ready for the PVS calculator to use.

Admittedly the above code snippet still leaves a number of important questions unanswered -- how do we create the initial portal on the plane to begin with? How do we perform that complex procedure of clipping portals to the tree? These topics will be explained in the next two sections.

17.5.3 Creating the Initial Portal

As we saw in the code in the previous section, if a node has both a front and a back child it has the potential to be a portal generator. When this is the case, we must create a huge initial portal on the node plane and then clip it to the tree. In the previous code this was assumed to take place in a function called CPortal::GeneratePortalFromPlane, so we will carry that example over into this section. The function took two parameters -- the plane on which to construct the portal and the bounding box describing the area that the portal must completely fill. The polygon generated may be larger than the box, but the function will always generate a polygon that at least fills it along that plane. The root node's bounding box was passed as this parameter, which means we always create an initial portal at a node that is large enough to span the entire level. This is important since the plane may have split geometry into two leaves right on the outer edges of the geometry set. If the portal is not big enough to begin with, it will never make it into the two leaves it is supposed to. This would mean that we would risk not generating enough portals and would lead to the calculation of incorrect PVS data.

So given a bounding volume and a plane, how do we create a polygon on that plane that is large enough to at least fill the volume? We first retrieve the center position of the box and classify it against the plane. The box center is shown as 'CB' in Figure 17.11. By classifying this point against the plane, we get the distance to that plane from CB along the plane normal. We can then add the negated plane normal multiplied by this distance to CB, thus moving CB along the normal until it rests on the plane. This is the projection of the box center onto the plane and is shown as CP in Figure 17.11. This will be the center of our portal. Out next task is to generate two tangent vectors for the plane which we can use to position the portal vertices on the plane. Because we have the plane normal, we can cross this with any vector that is not identical to generate vector U in the diagram. We can think of this as the right

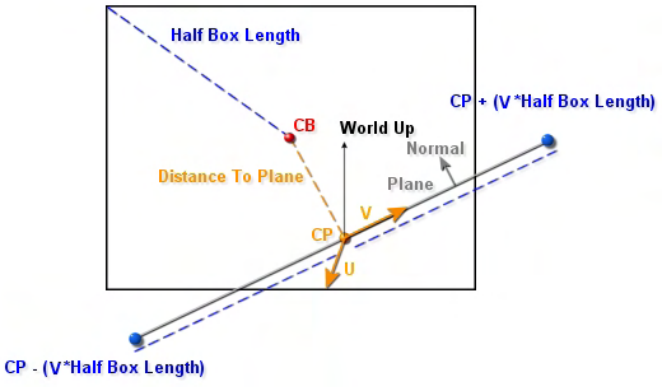


Figure 17.11

vector of the portal. In Figure 17.11 we can see that the normal was crossed with the world Y axis vector to generate vector U, which is perpendicular to both. We can then cross vector U with the plane normal vector to generate the second tangent vector (the V vector). These two vectors will be unit length at this point, but will describe the direction we would need to traversal from the portal center to reach the right and top edges of the portal quad we wish to build.

All we need to do now is find out how big we wish to make these vectors so that the top and right edges are defined as being as far (or further) than the box extents. This step is quite easy as we can simply use half the diagonal length vector of the box, shown in Figure 17.11 as the blue dashed line extended from the box center to one of the box corner. The half diagonal box vector describes the furthest you can possibly travel from the center point before piercing the extents of the box. If we multiply our U and V vectors by this amount we will describe the right and top edges of the portal as being at least far enough away such that they extend outside the box (the typical case) or on the box boundary. Either way, these vectors can now be combined and used to describe the position of the four corner vertices of the portal quad. As Figure 17.12 shows, because the U and V vectors are tangent vectors, they describe the top and right edges of the vertices on the plane. Therefore, we can combine them to find the positions of the four portal vertices.

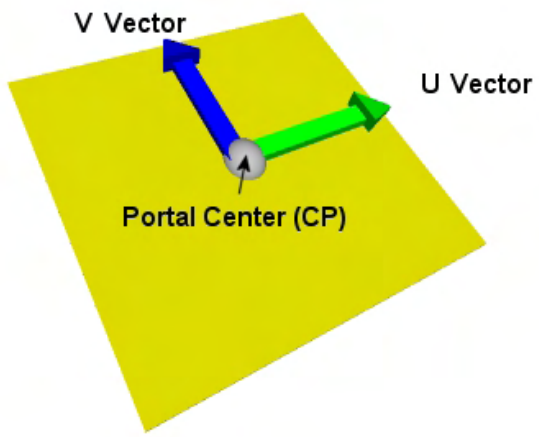


Figure 17.12

As CP is the center point of the polygon on the plane and U and V describe the half width and half height of the portal and are also tangent to the plane, we can generate the four vertex positions of the portal quad as shown below:

Bottom Right = CP + U - V;
Top Right = CP + U + V;
Top Left = CP - U + V;
Bottom Left = CP - U - V;

With this in mind, we now have everything we need to write a function that generates the initial portal. The first section is shown below. We calculate the center of the bounding box (CB) and then calculate the distance from this point to the plane. We then move CB along the plane normal by the negative of this distance to place the point on the plane. We store this in CP (short for Center of Portal).

```
bool CPolygon::GenerateFromPlane( const CPlane3& Plane, const CBounds& Bounds )
{
      CVector3 CB, CP, U, V, A;

      // Calculate BBOX Centre Point
      CB = (Bounds.Max + Bounds.Min) / 2;

      // Calculate the Distance from the centre of the bounding box to the plane
      float DistanceToPlane = DotProduct( Plane.Normal, CB) + Plane.Distance;
```

```
      // Calculate Centre of Plane
      CP = CB + (Plane.Normal * -DistanceToPlane );
```

Now that we have the portal center we need to create the U and V vectors. We first generate U by crossing the plane normal with an arbitrary vector which is non-identical. We choose this arbitrary vector from one of the world axes. Essentially, we choose a normal vector (A) which is least aligned with the normal of the plane, so that we get a cross product with plenty of resolution. That is, vector A will either be (1,0,0), (0,1,0) or (0,0,1) and we want the one that is least like the plane normal, just to be safe. As we know, if the two vectors fed into the cross product are identical (with tolerance), the cross product result will be undefined and we will not get back a vector that is perpendicular to the two input vectors (thus tangent to the plane).

```
      // Calculate Major Axis Vector
      A = CVector3(0.0f,0.0f,0.0f);

      if( fabs(Plane.Normal.y) > fabs(Plane.Normal.z) )
      {
            if( fabs(Plane.Normal.z)  < fabs(Plane.Normal.x) )
                A.z = 1;
            else
                A.x = 1;
      }
      else
      {
            if (fabs(Plane.Normal.y) <= fabs(Plane.Normal.x) )
                A.y = 1;
            else
                A.x = 1;
      } // End if
```

With vector A chosen, we cross it with the normal vector for the plane to generate the first tangent vector U. We then cross V with the plane normal to generate the second tangent vector V. Now we can normalize these vectors to make sure they are at all unit length.

```
      // Generate U and V vectors
    U = A.Cross(Plane.Normal);
    V = U.Cross(Plane.Normal);
    U.Normalize(); V.Normalize();
```

Finally, we subtract the center position of the box from maximum extents vector of the passed bounding box which gives us the diagonal half length vector of the box. We then fetch the length of this vector and use it to scale the lengths of the two tangent vectors. We then allocate an array of four vertices which will be used to store the four vertex positions of the quad and use U and V to calculate their positions as described above.

```
      float Length = GetVectorLength(Bounds.Max - CB);

      // Scale the UV Vectors up by half the BBOX Length
      U *= Length; V *= Length;
```

```
      CVector3 P[4];
      P[0] = CP + U - V; // Bottom Right
      P[1] = CP + U + V; // Top Right
      P[2] = CP - U + V; // Top Left
      P[3] = CP - U - V; // Bottom Left
```

We now have the four vertex positions, so we will increase the vertex array of the current portal being generated (remember that this function is assumed to be a method of the portal class in this example) and then copy the four vertex positions we have just calculated into those vertices.

```
    // Allocate new vertices
    if (AddVertices( 4 ) < 0) return false;

      // Place vertices in poly
      for ( int i = 0; i < 4; i++)
    {
        Vertices[i] = CVertex(P[i]);

    } // Next vertex

    // Success!
    return true;
}
```

So we have now seen the function that generates the initial portal for each node that has two children during the portal generation process. Once the initial portal has been generated for the current node being processed for portal candidacy, this initial portal is then sent into the ClipPortal function. This is the function that clips the portal to the tree and returns a list of surviving fragments which can be added to the master portal list.

17.5.4 Clipping the Initial Portal to the Tree

In this section we will spend a good amount of time covering the portal clipping process as this is where all of the hard work is done. Although it is essentially a CSG routine that clips a portal to a solid/empty tree, there are a lot of things to consider in all of the classification cases at each node.

Before we even begin to look at the code to such a function, we will look at some images that show the portal clipping process a step at a time. This will allow us to more easily digest all of the rules we discussed earlier for classifying the portal against each node in the tree.

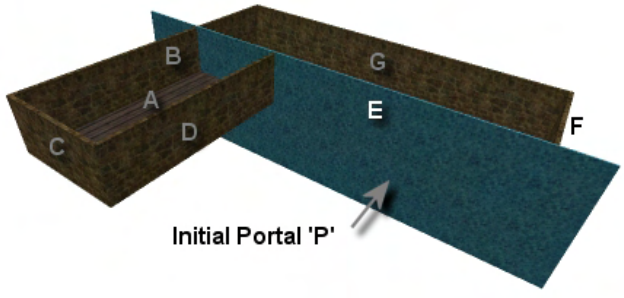


Figure 17.13

In this section we will use the geometry (and tree) that we showed being built in an earlier section (see Figure 17.9). When we examined the tree we saw that only node E had both front and back children and therefore, was the only node that could possibly generate a portal for our two leaf

level. This means that we would need to generate an initial portal on that plane that was large enough to span all the geometry. This initial portal (created by the function discussed in the previous section) is shown as the blue slab in Figure 17.13. In this image we have labeled the polygons themselves with the labels of the nodes that were created from those polygons.

We will now feed this initial portal ‘P’ into our BSP tree and do a dry run through the process that our ClipPortal function will need to perform on it. At the end of this section, you should have a much better understanding of why the portal traversal logic we discussed earlier works and generates correctly sized portals.

The large initial portal ‘P’ is passed into the root node and is classified against node plane A. This plane is shown in Figure 17.14 and is the floor plane of our level. The initial portal on plane E is spanning this plane so it is split into two child portals. The original portal is discarded and replaced with the two child portal fragments P1 and P2. P2 lies behind the plane, so is supposed to be passed down into the back child of node A. But as you can see, because there is no back child, it means that P2 must have ended up in solid space and thus we delete it. This makes sense since we can see in the image that portal P2 is clearly underneath the floor plane and thus would be situated in solid space.

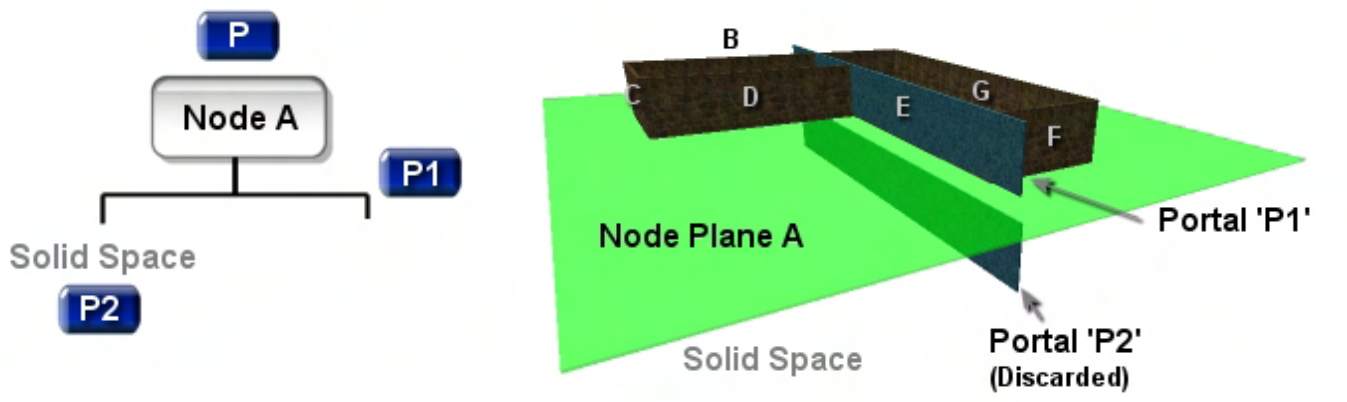


Figure 17.14

We have just performed our first split and the back split has been discarded. This is because we needed to pass it into the back child, but no back child exists. Again, we know that in a solid/empty tree this means it represents a solid area.

Our next task is to send the portal fragment P1 into the front child of node A because this portal fragment was the section that exists in the node’s frontspace. You are once again reminded that in all of these diagrams, the plane normals are assumed to face into the leaf. Therefore, in the case of the floor plane A, the normal is assumed to be pointing vertically up so that ‘in front’ equates to ‘above’, and ‘behind’ equates to ‘below’.

Node A does have a front child (node B) as shown in Figure 17.15. When portal P1 enters node B, it is found to be spanning the plane and is once again split into two child portals. The original portal P1 is deleted and is replaced with the two child split portals labeled P3 and P4 in Figure 17.15.

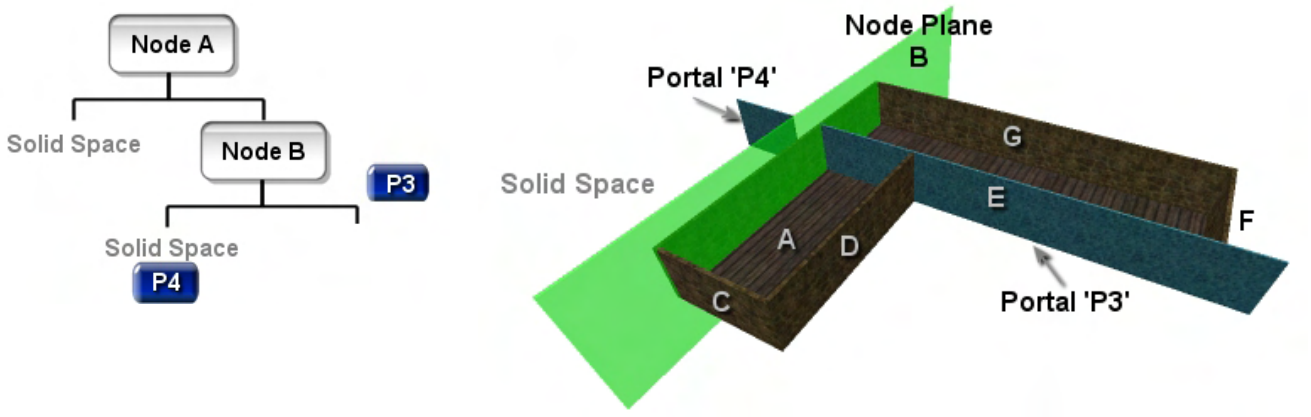


Figure 17.15

In Figure 17.15 we can see the section of the original portal (P1) that was split by node plane B into portal P4. Portal P4 is behind node B and as such would normally be sent into the back child. But since node B has no back child, it means portal P4 has ended up in solid space and should be deleted. Portal P3 however is contained in the frontspace of node B and as such should be passed into the front child. The front child of node B is node C. Therefore, portal fragment P3 becomes the input to node C.

When P3 is passed into node C is it found to exist in the frontspace of node C, as can be seen in Figure 17.16. This means it is passed into the front child of node C.

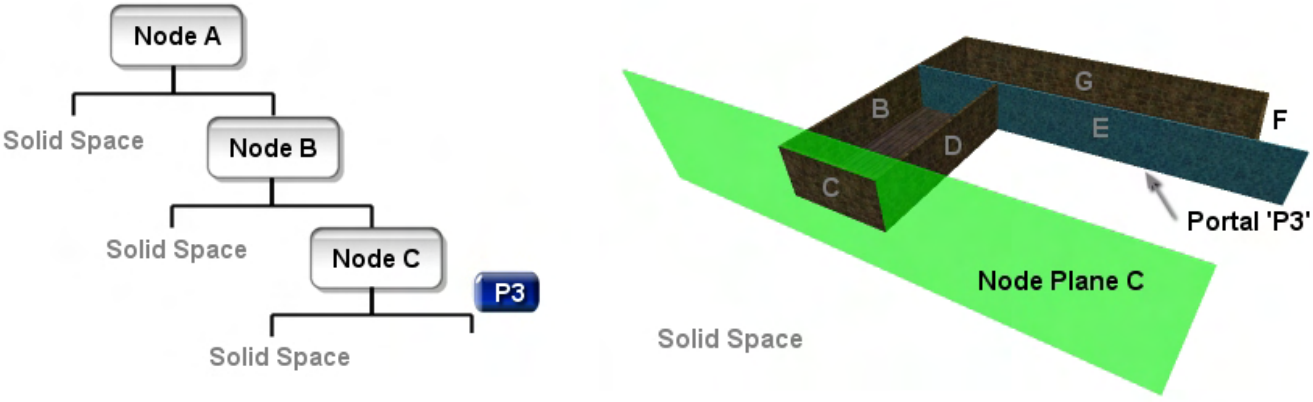


Figure 17.16

When we built this tree earlier you will hopefully recall that node E was selected next (instead of node D). Therefore, portal P3 becomes the input for node E and, since this portal was initially created on node E, it will obviously be found to be on plane.

As we discussed earlier, when the portal is found to lie on the node plane, we send the portal down into the front child first. If any fragments of the portal survive the front tree, they will be returned and passed down into the back child separately. Figure 17.17 shows the portal being classified against node E and it should be clear that the portal is located on the plane. This means that we must send portal P3 down E's front list first.

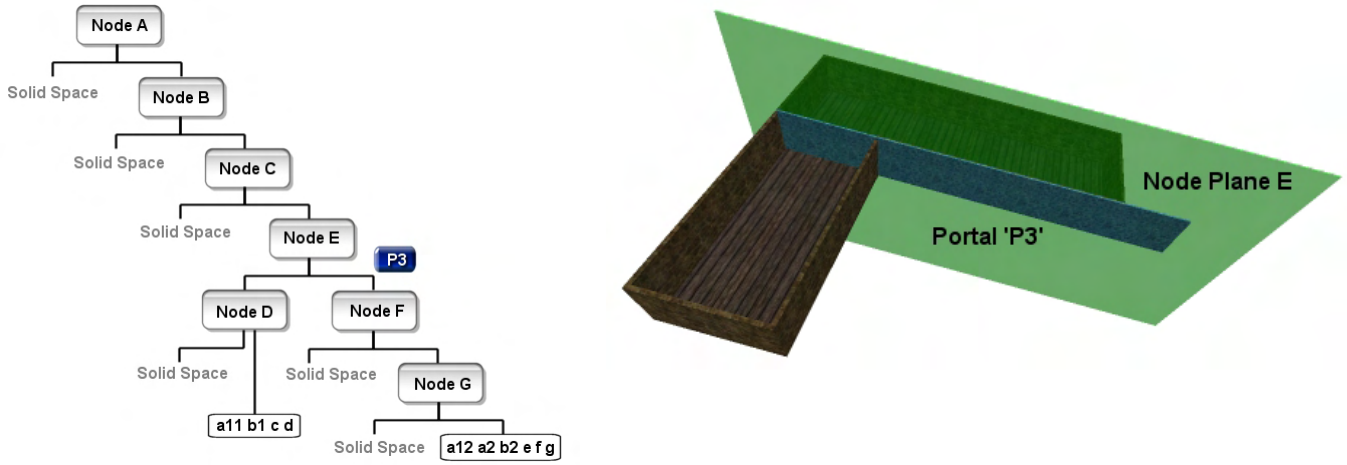


Figure 17.17

Let us now track portal P3 as it makes its way down the front sub-tree of node E. The normal at node plane of E is assumed to point into the top region of the geometry and thus is facing away from us in this image. This means that as we pass P3 down into the front child of node E, it enters node F. In Figure 17.18 we will show its path down the remaining branch of the node E's front tree.

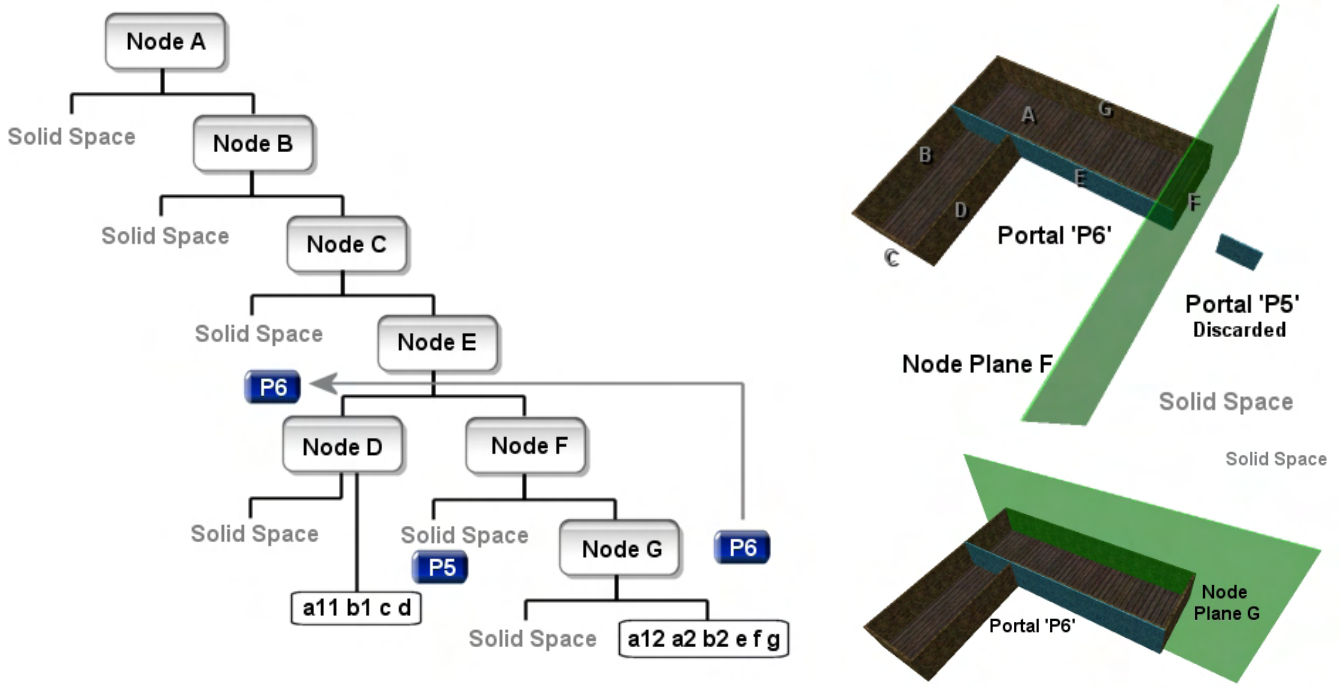


Figure 17.18

As we can see in the diagram, as P3 enters node F it is classified against the plane. It is found to be spanning plane F and is therefore split. The original portal P3 is deleted and replaced with the two child splits P5 and P6. P5 is situated in node F's backspace and enters solid space behind that node. This means portal P5 is deleted. Portal P6 is in node F's frontspace so is passed into the front child where it enters node G. Portal P6 is found to lay in the frontspace of node G as well, so it is passed down the

front child where it enters a leaf. The index of the leaf it has entered is retrieved from the tree (more on this later) and stored in the portal. Portal P6 is then returned back up through all the parent nodes until it is finally returned back to the on plane case at node E.

You will recall that because the portal was on plane with node E, it had to be sent down the front tree first and then any surviving fragments will be sent down the back tree. Portal P6 is the only surviving fragment from node E's front tree and has the index of the leaf it ended up in stored within its structure. Node E, having been returned portal P6 from its front list, must now send this portal down its back tree.

When we pass the portal P6 into the back child of node E, it enters node D. As can be seen in Figure 17.19, the portal P6 is found to be spanning node D and so it is split to the node plane. P6 is deleted to be replaced by the two split fragments P7 and P8. What is vitally important to note is that when this portal is split, the two child portals inherit the parent portal's leaf list. This means, if the leaf that portal P6 ended up in down the front tree of node E was leaf 64, both portals P7 and P8 would also have a leaf index of 64 stored in them also. This makes sense as these fragments were part of the parent portal when they landed in this leaf so they too exist in the leaf the parent did.

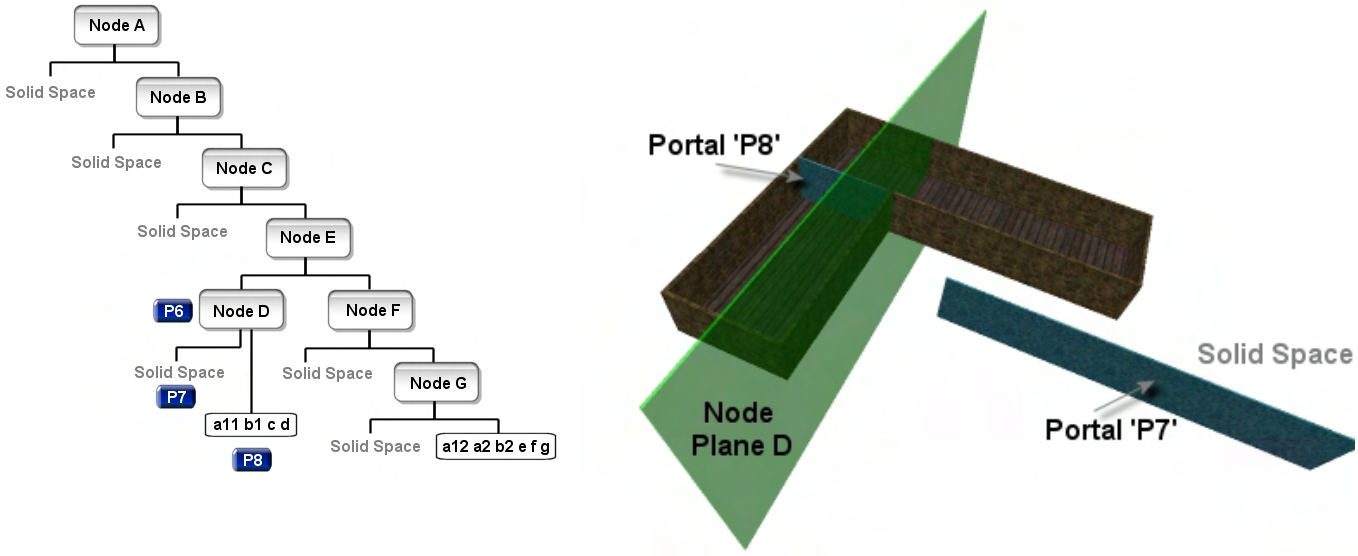


Figure 17.19

Portal P7 is found to lie behind node D and as such we identify it as existing in solid space. Portal P8 is found to lie in front of node D and is passed into the front child. However, node D is a terminal node which means anything that is passed down the front of it enters a leaf. The index of the leaf that portal P8 enters is added to its leaf list so that portal P8 now stores the indices of both the leaves it ended up in. Portal P8 is the only surviving fragment of the initial portal P that was fed into the root node. This portal is propagated up to the root node as the stack unwinds and is returned from the ClipPortal function to the parent process. The parent process is the ProcessPortals function we saw earlier which created the initial portal P on node E and sent it into the tree to be clipped. Portal P8 is returned from the ClipPortal function where it is added to the master portal list as a valid portal. Figure 17.20 shows the final shape of portal P8 and the two leaves it was found to exist in. As you can see, this portal exactly plugs up the gap in the geometry between leaf 1 and 2 and is therefore, the portal that leads from leaf 1 *into* leaf 2 and vice versa.

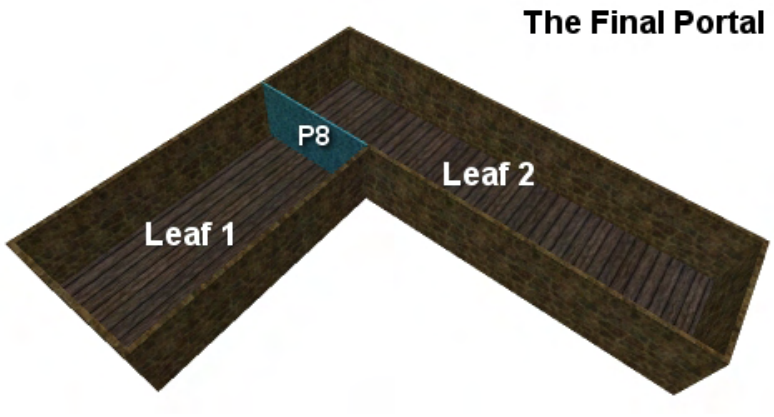


Figure 17.20

As we have now discovered, it is the solid/empty arrangement of geometry that makes this portal generation possible, as was the case with meshes used in CSG operations. If the geometry fed into the BSP compiler is not defined such that solid space is always bounded, we will no longer be able to tell which areas are solid and which are empty. This will obviously produce incorrect portals since the solid space tells us when a portal fragment needs to be discarded.

The above example that generated a portal on node E was a somewhat simple case in as much as it returned only a single portal. As we discussed earlier when reviewing the portal clipping process, a single initial portal may be clipped into many smaller valid portals and a linked list of these portal fragments may be returned from the clipper. Although this may sound a bit strange, it is actually quite obvious that this is the case when we consider that a node plane splits all the geometry in its space. Think of the root node as a perfect example. If the root node passed through the center of the scene for example, many rooms, passageways, etc. would by split into two pieces by this plane and thus, portals would be developed on those planes to bridge the split.

In the next example we will use slightly more complex geometry and a node plane that will create two portals. We will not build or show the BSP tree that has been compiled as it is not really necessary to get the idea of what we are trying to demonstrate.

Figure 17.21 shows a level comprised of two detached mesh constructs which we will now use to demonstrate a situation where a single initial portal can be clipped to the tree resulting in two valid portals being returned from the process.

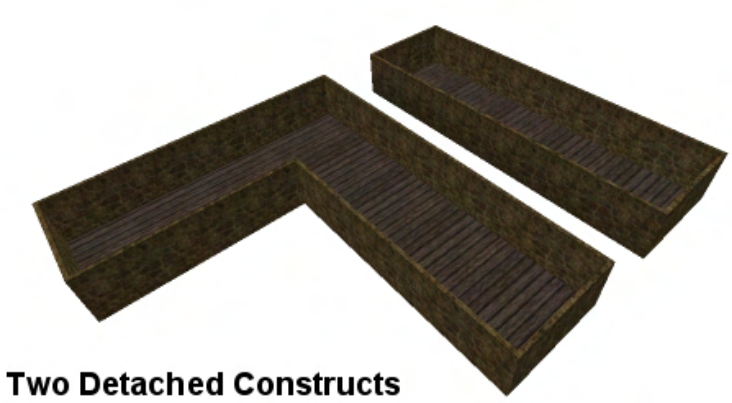


Figure 17.21

In Figure 17.22 we see the level again with the walls labeled alphabetically, describing the order in which the planes were placed in the tree. The initial portal we will create will be on node D which, in

this example, was responsible for splitting the two mesh constructs into two leaves when the BSP tree was compiled.

It should be noted that I and B are co-planar so would both be marked as having been used as a splitter when B is selected as the second node. Likewise, polygons F and K would only create one node (F) as they too are co-planar. The ProcessPortals function that we looked at earlier is assumed to have looped through each node in the tree and determined that node D was a node that had both front and back children and therefore, an initial portal would be created on that plane and clipped to the tree. It is this clipping, and the generation of the portals on that plane which we will walk through now.

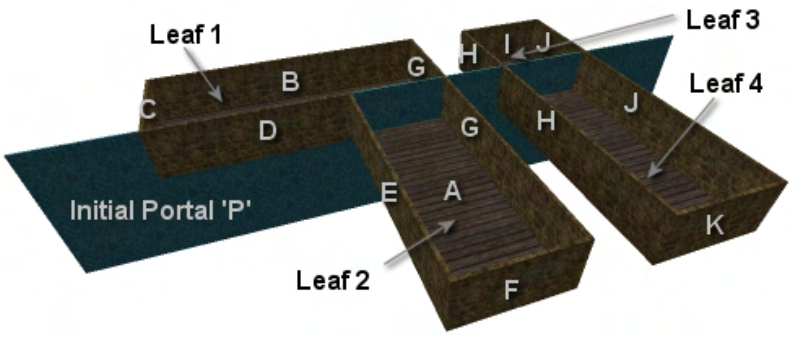


Figure 17.22

In Figure 17.22 we see the initial portal created on node plane D, which we have labeled portal P. Notice that it spans the entire geometry set of the level along that plane as all initial portals should do.

Portal P is first passed into the tree where it is classified against the plane of node A (Figure 17.23). In this example, node A is the floor plane and portal P is clearly spanning that plane, so it is split into portals P1 and P2. Portal P2 is sent down the back of A where it ends up in solid space and portal P1 is passed down the front tree where it enters the second node in the tree, node B.

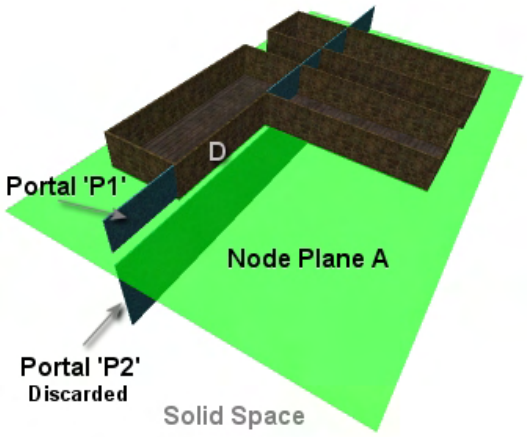


Figure 17.23

Figure 17.24 shows what happens when the surviving fragment portal P1 is passed into the front child of node A, which is node B. This is the plane of the polygons on the far walls of the construct and we can clearly see that portal P1 is in front of the node plane. This means portal P1 is passed into the front child of node B which, in this example, will be node C. Node C's plane is shown in figure 17.25.

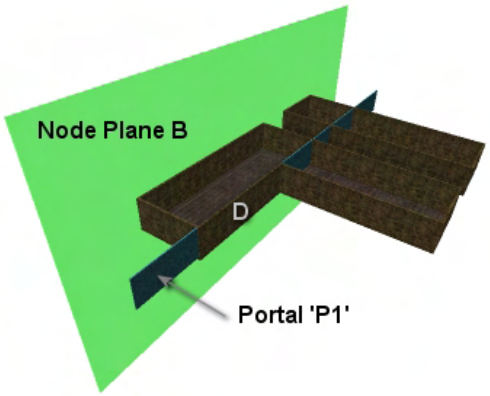


Figure 17.24

When portal P1 enters node C it is classified again the plane at that node. As we can see in Figure 17.25, it is found to be spanning the plane and is therefore split into two child portals (P2 and P3) and deleted from memory.

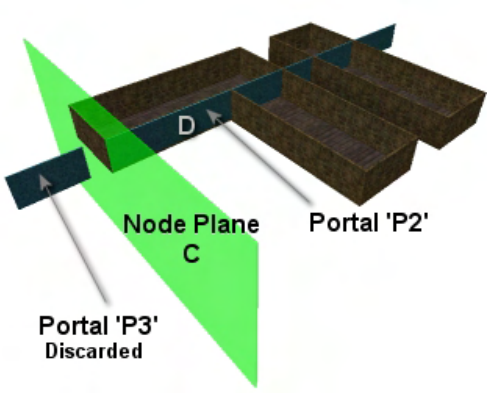


Figure 17.25

P3 is in the plane's backspace and is therefore passed down the back of node C where it is identified as being in solid space. Because of this, portal P3 is deleted as it could not possibly be a valid portal. Portal P2 however is contained in node C's frontspace and is passed down the front list of C where it arrives at node D. This is where things begin to get interesting, because node D is the node on which the initial portal was created. In this case, the portal is obviously going to be found to be co-planar.

As we know, when a portal is found to be situated on the current node plane, we must first send the portal down the front tree of the node. If any portal fragments survive the front tree they are each individually sent down the back tree.

Looking at Figure 17.26 we can see that node plane D was also the node that was responsible during the compilation process for dividing the one non-convex construct and the convex construct into four convex constructs. Remembering that the normals all face inwards, we can see in the diagram that the front child of D would be node G (a split plane which actually exists down both sides of D). Therefore when the portal P2 is passed down the front of D, it arrives at node G.

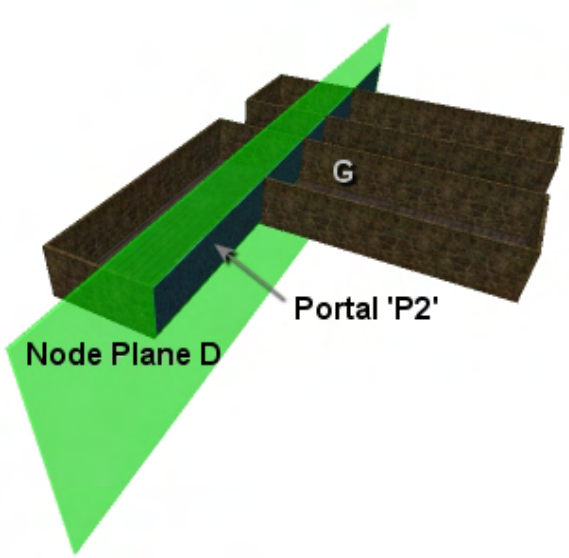


Figure 17.26

Notice where wall G is in the above diagram, because for the next image we will rotate the camera to a different view point so that we can see the portal clipping that is happening on the far side of the level. Also remember that when we are processing the planes down the front of D, any portal fragments that survive down that branch of the tree will be returned to node D and each will be passed down the back tree as well.

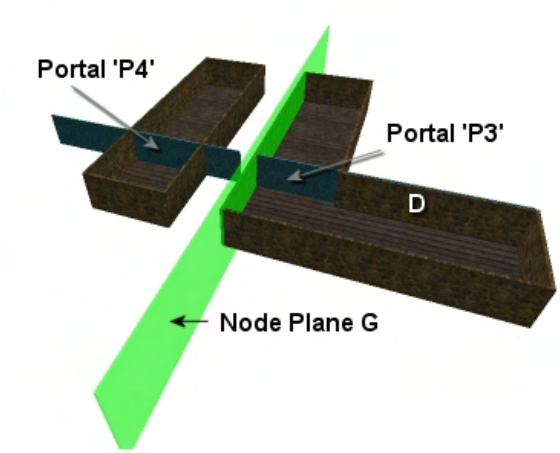


Figure 17.27

In Figure 17.27 you can see that down the front of node D the portal is then tested against node plane G where it is found to be spanning. Portal P2 is deleted and replaced by the front and back child splits P3 and P4, respectively.

P3 is in node G's front halfspace, so is passed into the front of the node where it ends up in a leaf. We can see this is the case as it ends up in the region bounding by all the planes we have already tested surrounding this area. The index of the leaf it has ended up in is recorded in portal P3 (which has so far ended up in one leaf).

Portal P4 is found to be located in node G's backspace, so is passed down the back where it is tested against node H. Portal P4 is found to be spanning node H, so is split into portal fragments P5 and P6 as shown in figure 17.28.

At this point we now have three portal fragments in existence. However, portal P5 is found to lay behind node H, which means it is in solid space. This describes it as existing in the space between the two constructs as shown in Figure 17.28. Because of this, portal fragment P5 is deleted.

Portal fragment P6 however is found to exist in the frontspace of node H and as such is passed into the front child where it moves into node J, as shown in figure 17.29.

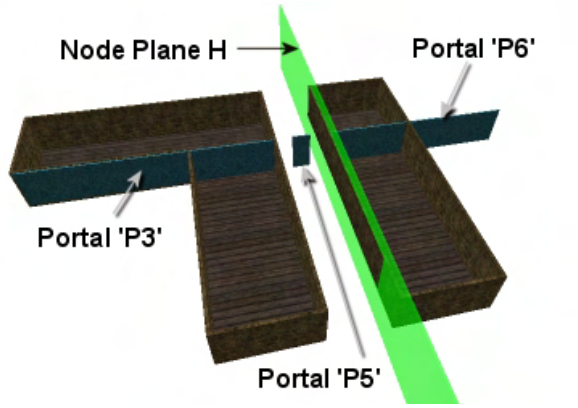


Figure 17.28

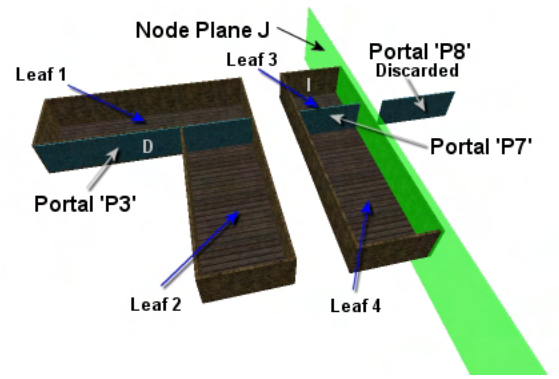


Figure 17.29

At node J, the portal P6 is spanning the plane and is split into portals P7 and P8. Portal P8 is in the back space of this node and is deleted because it ends up in solid space. Portal P7 is passed down the front, but as this is a terminal node, we know that this means it has entered a leaf (Leaf 3 in this example). The index of the leaf is stored in portal P7 and we have reached the end of the road down this branch of the tree.

Portal P7 is returned and the stack unwinds back up to plane G where the portal was originally split (see Figure 17.30).

Figure 17.30 depicts the situation after we have returned to node D, where the original portal was split and sent down the front and back branches of the tree. As discussed a little earlier, the front split was sent down the front where it ended up in Leaf 1 and the back split was send down the back where it got clipped a few more times before the fragment P7 eventually ended up in Leaf 3. This means that at node G, portal P3 would have been returned from the front tree and portal P7 would have been returned from the back tree. As discussed earlier, in the spanning case, the returned fragments from both of the nodes are joined together in a linked list and returned to the parent node. So when node G returns flow back to its parent node D, it returns two portal fragments from node D's front list.

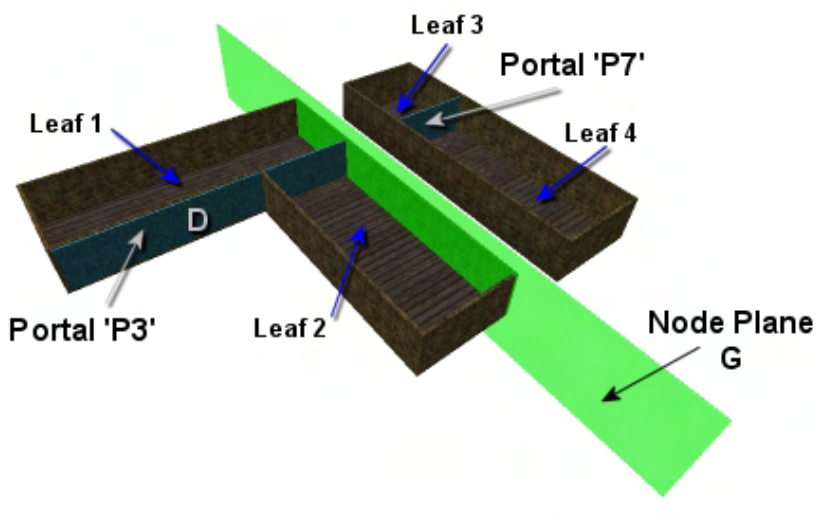


Figure 17.30

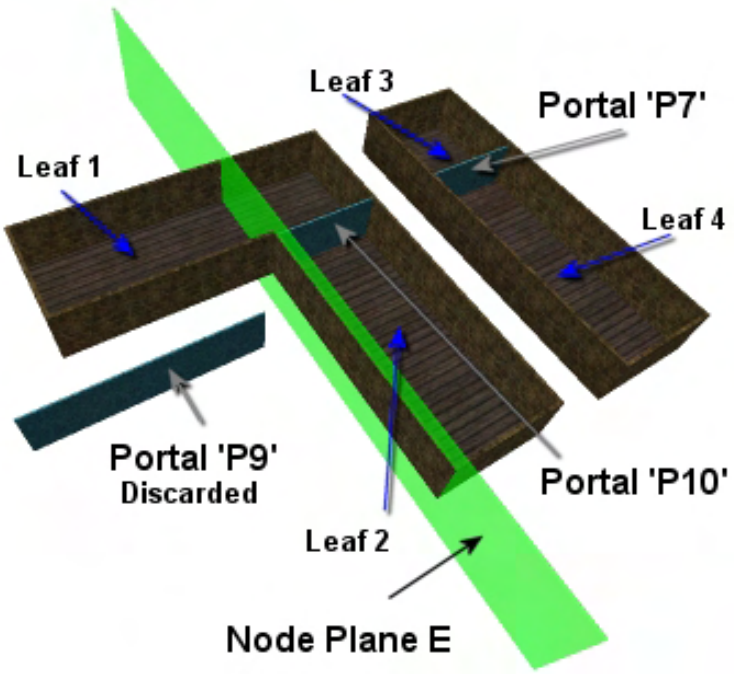


Figure 17.31

Node D was the plane on which the initial portal was created and therefore this was the plane on which the portal was found to be 'on plane'. We know that in the on plane case, the portal is first clipped to the front tree and any returned fragments sent down the back tree. We have now processed the front tree of node D and we have gotten back portals P3 and P7 which have been found to exist in leaves 1 and 3, respectively.

When portals P3 and P7 are returned to node D from its front list, each of these must now be clipped to the back tree as well. Although this is done one at a time in code we will show this happening simultaneously in the diagrams.

As Figure 17.31 shows, when P3 and P7 are passed down the back of node D they enter node E where they are classified against its plane. Portal P7 is completely contained in node E's frontspace, so is sent down the front tree as we would expect. Portal P3 however is found to be spanning node E so is split at the plane. Portal P3 is deleted and replaced by the child portals P9 and P10. Portal fragment P9 is passed down the back of E where it is found to exist in solid space and is therefore deleted. Portal P10 is

contained into the frontspace of node E and, like portal P7, is sent into the front tree. This means both portals P7 and P9 enter node F, as shown in Figure 17.32.

When portals P9 and P7 enter node F they are classified against the node plane and found to both be in front of that plane. Notice in the diagram how we also draw attention to the fact that this node plane is the plane that both polygons F and K lay on, and therefore a node would not have been created for both (K would have been marked as a splitter when node F was first selected). We can already see at this point that portals P9 and P7 are the final portal shapes we are looking for, but let us continue the process a little further just to make sure they both end up in two leaves down the back of D. Fragments P9 and P7 are both passed down the front of node F where they enter node G (see Figure 17.33).

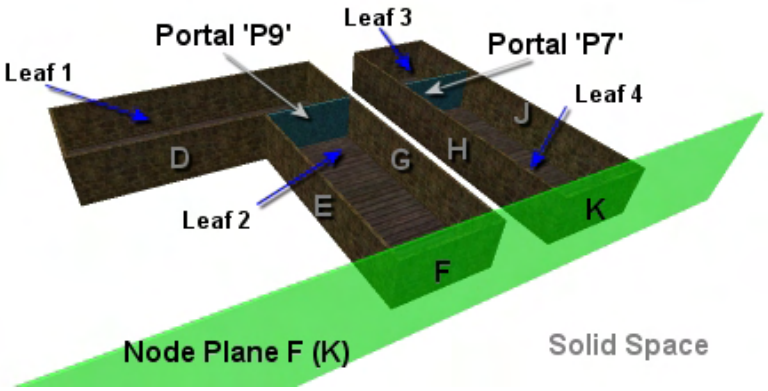


Figure 17.32

Notice that we are using some artistic license here when we refer to this node as node G. We already processed a node G down the front tree of plane D, and this is obviously not the same node. However, because polygon G would have been split during the build process by node plane D which was selected before it, its child splits of polygon G would have been passed down their relevant sides of the tree and used as splitters later. This means the split plane that we are referring to as G would in fact have been chosen as a node plane both in the front and back trees of node D.

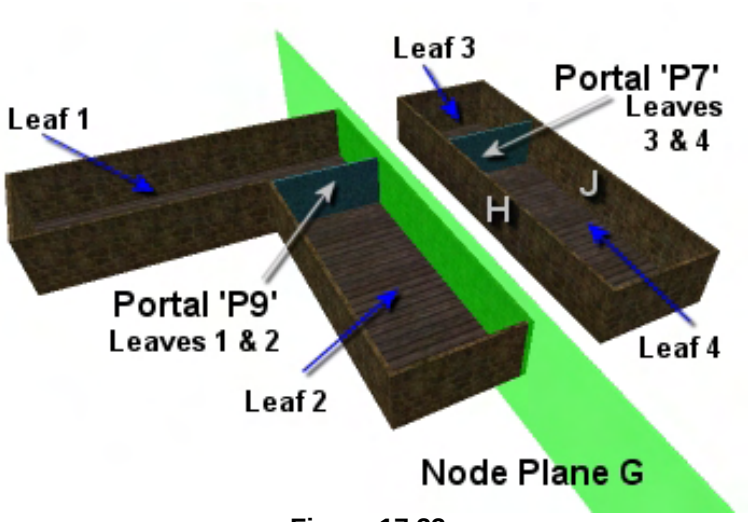


Figure 17.33

At node G, portal P9 is found to exist in the frontspace, so is passed into the front tree. However, Leaf 2 is attached to the front of node G and therefore portal fragment P9 has just entered its second leaf, indicating that it is a valid portal. The index of portal P2 is also added to the portals leaf owner array so that on function return this two element array will contain the two leaf indices in which the portal resides. This portal now exists in leaves 1 and 2 as can be seen in Figure 17.33. We will quickly describe the remaining path for portal P7, which is also nearly over.

Portal P7 is found to be in the backspace of G, which means it would be passed into the back tree where it would enter node H. When classified against node H it would be found to be in that node's front space and would be passed into its front tree where it enters the final node down that branch of the tree, node J.

When classified against node J it is found to exist in its front space and should be passed down the front, where it enters Leaf 4. This is also the second leaf that this portal has entered, and thus it is a valid portal. The index of the leaf is also added to the portal's leaf array which will then contain the leaf indices 3 and 4 in this example. At this point, the stack unwinds all the way back up to node D where it has had returned portals P7 and P9 from its back tree. These are stitched together into a linked list and returned from the node and the stack will unwind all the way back up to the root. These two portal fragments are exactly what will be returned from the ClipPortal method for the initial portal created on node D. They are ultimately the two portals that would be added to the master portal list, ready to be used by the PVS calculator.

Recursive Portal Splitting

As you might imagine, the portal clipping process utilizes a highly recursive algorithm. Although the code to the ClipPortal function in the lab project might look a little intimidating, much of this is the linked list management that must happen in the split case. Figure 17.25 shows an example of a split case and how the lists are maintained.

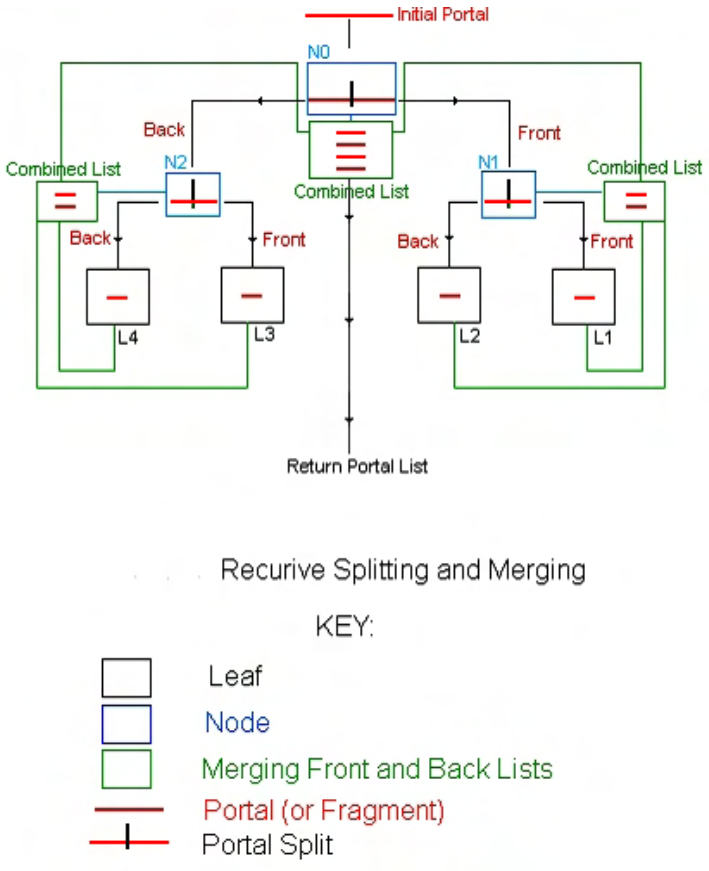


Figure 17.34

In this example, the initial portal is fed into the root node (N0) and is found to be spanning. It is split into two and the front fragment sent down the front into node (N1). At N1 this fragment is also found to be spanning so is split into two further fragments and sent down the front and back of N1 where it enters two leaf nodes (for the sake of this demonstration we are assuming back leaves as well). The portal fragments that made it into leaves L1 and L2 are returned to N1 from the front and back trees, respectively. They are then stitched together into a linked list at that node so that the two portal fragments can be returned to the parent node, N0. We can now see that node N0 has had a linked list of two portals returned from its front tree. However, we are still to process the back fragment that was split at node N0. This fragment is sent down the back tree where it enters node N2 and is further split. These two splits end up in leaves L3 and L4 down the front and back of this node. That is, N2 will have portal fragments returned from both its front and back trees, which are then stitched together and returned to the root node. At the root node N0 we can now see that it has two portals returned from its front tree and another two returned from its back tree, making four valid portal

fragments in total. These two lists are then stitched together into a single linked list of four fragments which is then returned from the function back to the calling process.

Leaf Classification

When a portal enters a leaf we will store the index of that leaf in the portal's two element leaf array. As we now know, if the portal is a valid portal then it will eventually have this array filled with the two leaf indices in which it was located. However, you will see later that we do not just need to know in which leaves the portal is contained; we also need to know which leaf is behind the portal and which leaf is in front of the portal (with respect to the portal's normal). This will be important later when we calculate the PVS so that we know the direction in which the visibility calculations should flow from one leaf to the next. The determination will be done during the clip portal process at the time the portal enters the leaf. That is, we will classify the leaf against the portal and if the leaf the portal has just landed in is in front of the portal (i.e., the portal's normal is facing into that leaf) we will store its index in element zero of the portal's leaf array. If the leaf is found to be behind the portal, then the leaf index is stored in element one of the portal's leaf array. To make this easier to remember, we will set three defines in our code which we can use as the indices when we wish to fetch the leaf indices at the portal.

```
#define FRONT_OWNER          0
#define BACK_OWNER           1
#define NO_OWNER             2
```

As you can see these defines just describe the location in the portal's leaf array where the indices of front and back leaves should be stored. The NO_OWNER define will be explained in a moment.

One might imagine that classifying the leaf against the portal could be done by simply taking the center of the leaf's bounding box and classifying this point against the portal. However, this will not suffice as it is entirely possible that the leaf will contain a single polygon. It is quite possible due to some split planes chosen further up the tree that a portal might be created where the floor meets the wall within a room. Figure 17.35 shows an example of this. As you can see, all of the room except for the left wall is part of leaf, 1 but due do an odd split somewhere up the tree, the left wall polygon has fallen into leaf 2. Therefore, the portal that bridges these two leaves will actually exist on the node plane and worse still, because leaf 2 contains a single polygon, it will have a zero volume bounding box. Finding the center of leaf 2's box would therefore give us a point that is on the plane of the portal and

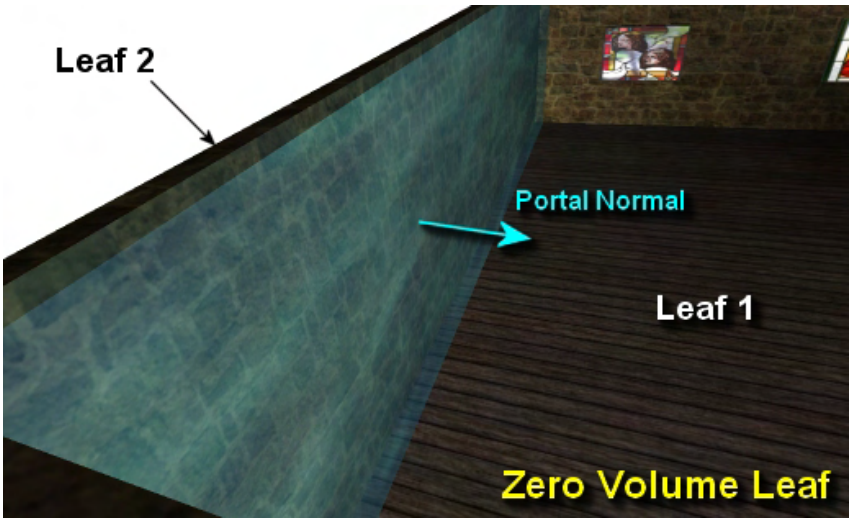


Figure 17.35

therefore we have no way of determining which side of the portal the leaf is on. Of course, the portal normal indicates that leaf 2 should exist behind the portal and leaf 1 in front.

In order to solve this problem and correctly identify on which side of a portal a leaf is located, when the portal enters that leaf, we will use a function called ClassifyLeaf to leverage the hierarchical nature of the BSP tree. Essentially it will locate which sub-tree of the node that was used to create the portal contains the leaf which the portal has just entered.

Some placeholder code for a ClassifyLeaf function is shown below. Is called from the portal clipper at the time a portal fragment enters a leaf. The function is passed the node on which the portal was created and the index of the leaf we are looking for (down either its front or back tree). We know that if this portal has ended up in a valid leaf, then it must be a portal that is either down the front or back tree of the node. After all, it is the node itself which split the geometry into two leaves and therefore, those leaves must exist below it in the tree. If the node is found down the owner node's front tree, then the function returns FRONT_OWNER which tells the portal clipper that the leaf is in front of the portal and should be stored in element 0 in the portal's leaf index array. If the leaf is found down the back tree, then it returns BACK_OWNER and the portal clipper function knows that it should store the leaf index in element 1 in the portal's leaf array.

> **Note:** In this version of the function it is assumed that if the front child index of a node is a negative number, then a leaf exists in front of the node and it is not just another internal node. The negative index has one added to it and is negated to get the index of the leaf in the leaf array that is stored there. This is in keeping with the way our data structures were arranged in the previous lab project when we first developed the leaf tree compiler.

The function is passed the index of the leaf the portal has ended up in and the index of the node (in the node array) that was used to create the portal. Now you know why we stored the index of the node that created the initial portal in the portal's OwnerNode member. When the portal lands in a leaf, we now have immediate access to this node and we can pass it into this function as the second parameter.

```
unsigned long ClassifyLeaf( unsigned long Leaf, unsigned long Node )
{
    CBSPNode * CurrentNode = NULL;

    // fetch the node pointer from the tree
    CurrentNode = pTree->GetNode( Node );
```

The function first uses the passed node index to fetch the node pointer from the tree's node array, as can be seen in the above code.

Our next task it to test to see if the node's front child index is negative. If it is, then the node has no front child node. Instead, the index stored in the node's Front member will be the negative index (minus 1) of the leaf that is attached to the front of that node. By taking the absolute value of the front index and adding one, we get the index of the leaf that is stored there. If this matches the leaf index that was passed into the function (i.e., the leaf we are trying to classify against the portal) then we have found it and we know the passed leaf is attached to the front of the node. In this case, we return FRONT_OWNER.

```
    // Check to see if the front is this leaf
    if ( CurrentNode->Front < 0 )
    {
        if ( abs(CurrentNode->Front + 1 == Leaf ) return FRONT_OWNER;
    }
```

If the front member of the node is not a negative number, then it stores the index of the front child node of the current node. This means we will need to recur down the front of the node and search for a matching leaf index down that branch of the tree. This is done via a function called FindLeaf, which we will cover in a moment.

```
    else
    {
        if (FindLeaf( Leaf, CurrentNode->Front )) return FRONT_OWNER;
    } // End If
```

As you can see in the above code, we pass this function the index of the leaf we are searching for and the node we wish to start searching from. This is the front child node of the portal's owner node. If this function returns true, it means that it located a matching leaf down the front branch of the node and therefore we know that the leaf exists to the front of the portal. Thus, we return FRONT_OWNER.

If we have not returned from the function at this point, then it means we did not locate the leaf down the front tree of the portal's owner node so we must now search to see if we can locate it down the back of the node. If the owner node has a back child then its Back index will be non-negative and will describe the index of the child node. Therefore, we pass this node index into the FindLeaf function, along with the leaf index we are searching for, to perform a search down the back tree. If the FindLeaf function returns true, it means that the leaf was located in the search and therefore, the leaf is situated behind the portal. Therefore, we return BACK_OWNER.

```
    if ( CurrentNode->Back >= 0 )
    {
        if (FindLeaf(Leaf, CurrentNode->Back )) return BACK_OWNER;
    } // End If

    return NO_OWNER;
}
```

Notice at the end of the function we return NO_OWNER if the portal leaf could not be found down either the front or the back tree of the portal's owner node.

At first, it would seem that this situation could never arise. After all, the reason we called this function is that the portal ended up in a leaf. So we can assume that leaf must exist somewhere underneath the portal's owner node, right? In fact, no. Returning NO_OWNER will inform the portal clipper that the portal has ended up in a leaf that it has no business being in and that the portal should be deleted. This may sound absurd, but we will soon examine how these rogue portal fragments can come into being. Just know for now that by returning NO_OWNER if the portal has not fallen into a leaf that exists beneath the owner, the portal clipper knows it has a rogue fragment that can be deleted. This completely removes the problem.

Before we examine rogue portals in more detail, let us take a look at the FindLeaf function called from the ClassifyLeaf function. It is a simple recursive procedure that walks the tree starting from the input node and searches for a matching leaf index. If one is found, it returns true.

```
bool CProcessPRT::FindLeaf( unsigned long Leaf, unsigned long Node )
{
    CBSPNode * CurrentNode = NULL;

    // Validate Requirements
    CurrentNode = m_pTree->GetNode( Node );

    // Check to see if the front is this leaf
    if ( CurrentNode->Front < 0 )
    {
        if ( abs(CurrentNode->Front + 1) == Leaf ) return true;
    }
    else
    {
        if (FindLeaf( Leaf, CurrentNode->Front )) return true;
    } // End If

    // Iterate down the back if it's a node
    if ( CurrentNode->Back >= 0 )
    {
        if (FindLeaf( Leaf, CurrentNode->Back )) return true;
    } // End If

    return false;
}
```

As you can see, the function recursively calls itself searching for leaf at the front of each node it visits. If a node is reached that has a negative front node index, then it means that this is actually the index of a leaf that is stored here. The index is transformed into a valid leaf index (by adding 1 and negating) and is compared against the leaf index passed into the function. If a match is found, it returns true. If a match is not found, it recursively calls itself to walk down the front or back of the current node if those children exist. The only time the end of the function is reached and false is returned is when the front and back trees of the node have been traversed and no matching leaf was found.

Rogue Portal Fragments

It may at first seem that if a portal ever ends up in a leaf, we would always want to keep it because it is in empty space. However, when we are trying to generate a portal on that node, remember that we are only interested in finding the two leaves that the node divides. As it so happens, there may be other nodes in the tree that share the same plane and as such would have initial portals built on the same plane. As all initial portals are fed into the root node of the tree, this means that if action was not taken as described above, some portal fragments from the node we are currently trying to generate portals for can end up in leaves that are not underneath that node in the tree.

To understand this scenario, take a look at Figure 17.36. In this example, we will imagine that the wall labeled Node A was chosen as the root node when the BSP tree was first compiled. We know that polygons G1 and G2 are on opposite sides of the plane and therefore would be passed down different sides of node A. We will also imagine that G1 was passed down the front tree of A where it was then selected as the split plane. Wall G2 is assumed to have been passed down the back of node A where it will be used as the next node plane down the back tree of A. We now have two nodes that have identical planes, but exist in different sub-trees of node A. This situation arises when a polygon is split and passed down into the front and back trees of the node *before* it has been selected as a splitter. Remember that each of those polygon fragments will later be used as a node plane.

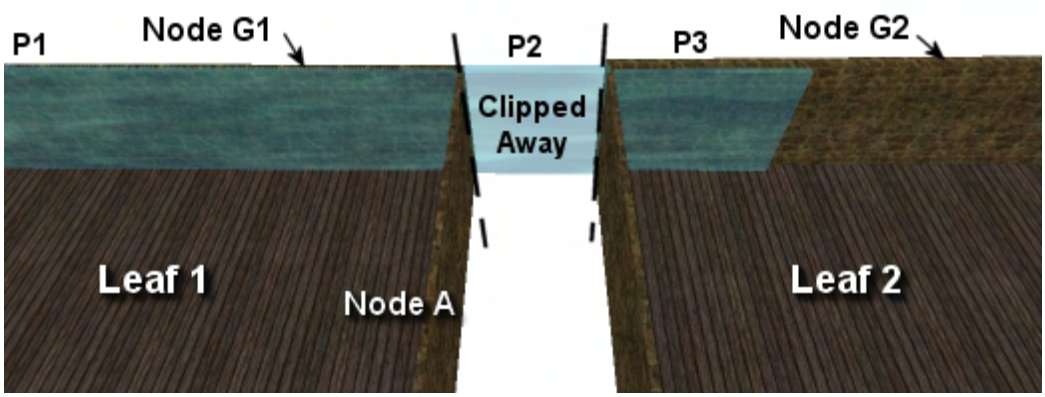


Figure 17.36

Now imagine that the tree is compiled and we are trying to find portal P. For this example we will say that the far walls of leaves 1 and 2 are not actually part of those leaves. Instead they exist in other leaves and as such, a portal will need to be created there. As we know, during the portal generation process we would determine that node G1 has two child nodes, so we should create an initial portal and send it down the tree. Of course, all we are interesting in finding is the fragment of the portal P1 which exists in leaf 2 and the leaf behind the portal. Initially, the portal P would be generated as a very large polygon and passed down the root node. Portal P is assumed to be the entire width of the blue portal shown in Figure 17.36. When this portal enters the root node, it is classified against node A and is split. The front fragment is sent down the front tree of node A where it ends up in leaf 1. This is the portal we were looking for, labeled P1 in the image. However, when the split occurred at node A there was also a significant portion of the portal that lay behind node A, and as such would be passed down the back list. Let us assume that it next enters a node where the split plane used is the left wall of the right construct. As we can see, this plane would split the fragment into portals P2 and P3. P2 would be sent down the back and deleted because it ended up in solid space. Portal P3 however is passed down the front where it ends up in leaf 2. So part of the portal that we were trying to generate for node 1 to plug the gap in leaf 1 has ended up in leaf 2. Furthermore, depending on the clipping process applied to that portal on its way to leaf 2, the portal fragment that actually lands there could be arbitrarily sized. Essentially, we want to get rid of this portal fragment (P3) since the portal that should be created here will ultimately occur when the initial portal is built on node G2. So how do we know if a portal ends up in a leaf that it is a rogue fragment that should be deleted?

Luckily, it is so simple that we have already taken care of it. The only way we can ever have two nodes sharing the same plane (such as G1 and G2 in the above diagram) is if they exist in different parts of the tree. This is due to the fact that the polygons used to create those nodes were separated into front and

back lists before either of them was used as a splitter. Therefore, the leaves we are looking for when generating a portal are always the leaves that exist below the portal's owner node in the tree. In Figure 17.36 for example, we can instantly tell that node G2 (and therefore leaf 2) would not exist underneath node G1 as one of its children. Therefore, when a portal enters a leaf, if we cannot find that leaf in either the front or back tree of the owner node, we know that we have a rogue fragment and that we can return NO_OWNER from our leaf classification function. The portal clipper can then delete this rogue fragment.

So as you can see, the classify leaf function solves two problems. First, it allow us to determine which portal halfspace the leaf in which that portal has just entered resides. Second, it also allows us to remove rogue portal fragments by testing whether the leaf that the portal has entered exists beneath the portal's owner node in the tree.

With a good deal of discussion, examples, and the examination of support functions under our belt, we are now ready to look at our first version of the portal clipper. It is a single recursive function that is called by the ProcessPortals function for each node that has an initial portal generated for it. Again, you are reminded that this is every node that has a front and a back child (or a back child and a front leaf).

The Portal Clipper

In this section we will examine the code to the recursive ClipPortal function, which is invoked to clip the initial portal generated at each applicable node (i.e., has two children). This code is only for demonstration and discussion and does not match up with all of the namespaces and types used in the actual lab project that accompanies this lesson (the actual source code will be discussed in the workbook). However, this version is very close to the true implementation and will provide a full understanding of the process we have been discussing thus far.

The ClipPortal function is called by the ProcessPortals function we looked at earlier. It is passed the root node and the initial portal that will need to be clipped to the tree. The passed portal will also store (in its OwnerNode member) the index of the node in the tree on which the initial portal was created. The function returns a pointer to a portal structure. Since this portal structure can be linked with other portal structures (via its Next member) the function may thus return a linked list of multiple portal fragments that survived being clipped to the tree.

The first thing the function does is use the passed node index to get a pointer to the node currently being visited by the function. This will be the root node the first time it is called from the parent process. When the node pointer is fetched from the tree's node array, its Plane member (which contains the index of the split plane stored at that node) is used to retrieve the relevant plane from the tree's plane array. The Portal is then classified against the plane. It is assumed in this code that the plane class has a ClassifyPoly method which classifies the portal polygon against the plane and returns a front, back, spanning, or on-plane result.

```
CPortal * ClipPortal( unsigned long Node, CPortal * pPortal )
{
```

```
    CPortal   * PortalList     = NULL, *FrontPortalList = NULL;
    CPortal   * BackPortalList = NULL, *Iterator        = NULL;
    CPortal   * FrontSplit     = NULL, *BackSplit       = NULL;
    CBSPNode    * CurrentNode    = NULL;
    CPlane      * CurrentPlane   = NULL;
    unsigned long  OwnerPos, LeafIndex;

    // Validate Requirements
    if (!pPortal || !m_pTree) throw BCERR_INVALIDPARAMS;

    // Store node for quick access
    CurrentNode  = m_pTree->GetNode( Node );
    CurrentPlane = m_pTree->GetPlane(CurrentNode->Plane);

    // Classify the portal against this nodes plane
      switch (CurrentPlane->ClassifyPoly(pPortal->Vertices,
                                         pPortal->VertexCount,
                                         sizeof(CVertex)))
    {
```

The CPlane::ClassifyPoly method parameters are a pointer to the polygon's vertices, the number of vertices in the buffer, and the size of each vertex structure. Our CPortal structure is a class that is derived from our CPolygon structure. Although it adds a few new members of its own, because it is a CPolygon, it can be passed to all functions that expect to work with polygon data. Notice in the above code that the result from the portal/plane classification is passed into a switch statement. The action taken on each result type will now be covered one at a time.

CP_FRONT

When the portal is found to be in front of the current node plane it means that we either have to send it down into the front child node if one exists, or if not, it means a leaf is attached to the front of the node and the portal has successfully made it into this leaf. We will cover the code that is executed if the portal has made it into a leaf first.

```
      case CP_FRONT:

            if (CurrentNode->Front < 0 )
            {
                // The front is a Leaf, determine which side of the node it fell
                LeafIndex   = abs(CurrentNode->Front + 1);
                OwnerPos    = ClassifyLeaf( LeafIndex, pPortal->OwnerNode );

                // Found the leaf below?
                if ( OwnerPos != NO_OWNER )
                {
                    // This is just returned straight away, it's in an empty leaf
                    pPortal->LeafOwner[OwnerPos] = LeafIndex;
                    pPortal->NextPortal          = NULL;
                    pPortal->LeafCount++;
                    return pPortal;
                } // End if leaf found
                else
                {
                    delete pPortal;
```

```
                    return NULL;

                } // End if leaf not found
            } // End if child leaf
```

In the IN_FRONT case we first test the value of the current node's front index. If it is negative then it means that it does not contain the index of a child node, but contains the index of a leaf structure that is attached to this node. As such, we know that the portal has made it into this leaf. The front index is negated and incremented so that it now represents the index of the leaf in the tree's leaf array, and then this index is passed into the ClassifyLeaf function that we covered in the previous section. The result of this function will be stored in the OwnerPos local variable and provided that it does not equal NO_OWNER, OwnerPos will be a value of either zero or one describing where in the portal's LeafOwner the index of the leaf should be stored. The portal's LeafCount member is also increased so that it correctly reflects how many (of the two possible) leaves have been added to this portal. For any valid portal that is returned from the function, this should be set to exactly 2. The portal's NextPortal member is also set to NULL since this is a single portal that is not yet attached to any linked list. This portal is then returned back to the parent node because it has survived clipping and made it into a valid leaf.

Notice in the above code however that if the ClassifyLeaf function returns NO_OWNER, it means that this leaf does not exist underneath the portal's owner node in the tree. Therefore, this is a rogue fragment and it should be deleted. After it is deleted, NULL is returned which indicates that the portal passed into this function did not survive the current node traversal.

The above section of code is executed when a leaf is in front of the current node being visited. However, if a leaf does not exist down the front of this node, then the node's Front index will contain the index of the child node. When this is the case, the function recursively calls itself, passing the portal down to the child.

```
            else
            {
                // Pass down the front
                PortalList = ClipPortal(CurrentNode->Front, pPortal);
                return PortalList;

            } // End If child node

            break;
```

Notice how the portal (or portal list) returned from the child is assigned to the PortalList variable and it is this list that is returned. Remember, although we are just passing the portal down the front of the tree, it may end up getting split into many valid fragments (or none, in which case NULL would be returned) down that section of the tree. The pointer returned to the child may point to the first portal in a linked list of many fragments that survived the front tree of the current node. It is this list that should be returned to the parent. That is the CP_FRONT case taken care of.

CP_BACK

If the portal is found to be behind the current plane then we first test the current node's Back child node index. If it is a positive number then it will describe the index of the back child node. If it is a negative number then it means that there is a leaf attached to the back of this node. However, if it is a negative number equal to BSP_SOLID_LEAF, it means that the portal, if passed down the back of this node would end up solid space. In this case, the portal is deleted and NULL is returned.

```
        case CP_BACK:

             // Test the contents of the back child
             if (CurrentNode->Back == BSP_SOLID_LEAF )
             {
                 // Destroy the portal
                 delete pPortal;
                 return NULL;

             } // End if solid leaf
```

The next section of code is executed if there is not solid space behind the node, but is instead a valid back leaf. This code should look very familiar. The Back child index is negated and incremented, transforming it into a valid leaf index (into the tree's leaf array). The leaf index and the portal's owner node are then passed into the ClassifyLeaf function and the result indicates whether the leaf is situated in front or behind the node. Provided the result was not NO_OWNER, the leaf index is stored in the relevant position in the portal's leaf index array and the portal's leaf count is incremented before the portal is finally returned from the function. If the result of ClassifyLeaf is NO_OWNER however, this means the leaf that the portal has made it into does not exist under the portal's owner node in the tree and as such, this is a rogue fragment that should be deleted.

```
             else if (CurrentNode->Back < 0 )
             {
                 // The back is a Leaf, determine which side of the node it fell
                 LeafIndex   = abs(CurrentNode->Back + 1);
                 OwnerPos    = ClassifyLeaf( LeafIndex, pPortal->OwnerNode );

                 // Found the leaf below?
                 if ( OwnerPos != NO_OWNER )
                 {
                     // This is just returned straight away, it's in an empty leaf
                     pPortal->LeafOwner[OwnerPos] = LeafIndex;
                     pPortal->NextPortal          = NULL;
                     pPortal->LeafCount++;
                     return pPortal;

                 } // End if leaf found
                 else
                 {
                     delete pPortal;
                     return NULL;

                 } // End if leaf not found

             } // End if child leaf
```

If the Back child index of the current node is non-negative, it signals that there is another child node attached to the back of this node and as such, we send the portal down the back tree.

```
            else
            {
                // Pass down the back
                PortalList = ClipPortal(CurrentNode->Back, pPortal);
                return PortalList;

            } // End If child node

            break;
```

The list of portal fragments returned (if any) from the back tree is then returned, as can be seen in the else conditional in the above code.

We have now covered the two simplest cases in the clipping procedure, the front and back cases.

CP_ONPLANE

The on plane case is quite a bit larger than the previous two but is not as complex as it first appears. The portal needs to be clipped to the front tree first and any surviving fragments then each need to be clipped to the back tree. The result is a list of fragments that have survived both the front and back trees of the node, which is then returned from the function. Of course, if a leaf exists to the immediate front or back of the current node, then the leaf index is simply recorded in the portal's leaf array as we have seen in the previous two cases.

In the first section of the code we test the front to see if it is the index of a leaf (i.e., it is negative). If it is, then the leaf index is passed into the ClassifyLeaf function as before and if found to be either behind or in front of the portal, it is added to the portal's leaf list. Notice in this case however that we do not simply return the portal when it is found to be in a leaf; it must also be passed down the back of the node. So instead, we cache the portal pointer in the FrontPortalList local variable.

```
          case CP_ONPLANE:

            if (CurrentNode->Front < 0 )
            {
                // The Front is a Leaf, determine which side of the node it fell
                LeafIndex   = abs(CurrentNode->Front + 1);
                OwnerPos    = ClassifyLeaf( LeafIndex, pPortal->OwnerNode );

                // Found the leaf below?
                if ( OwnerPos != NO_OWNER)
                {
                    // This portal is added straight to the front list
                    pPortal->LeafOwner[OwnerPos]= LeafIndex;
                    pPortal->NextPortal         = NULL;
                    FrontPortalList             = pPortal;
                    pPortal->LeafCount++;

                } // End if leaf found
```

```
                else
                {
                    delete pPortal;
                    return NULL;

                } // End If no leaf found

            } // End if child leaf
```

If the front child index is not a negative number then it means that a child node exists down the front of the current node. As such, the portal should be clipped to the front sub-tree and the address of the returned linked list of fragments should be stored in FrontPortalList .

```
            else
            {
                  FrontPortalList = ClipPortal(CurrentNode->Front, pPortal);

            } // End If child node
```

At this point FrontPortalList will either contain the portal that made it into the immediate child leaf down the front of this node or it will contain the linked list of fragments that survived the front tree. It might also be NULL if no fragments survived the front tree. If this is the case, we return NULL as shown below.

```
            // If nothing survived return here.
            if (FrontPortalList == NULL) return NULL;
```

We must return NULL if FrontPortalList equals NULL because if nothing survived down the front of the tree it cannot possibly be a valid portal (a valid portal is a portal that has a leaf on both sides of its plane). However, if the current node (with which the portal is on-plane) has solid space behind it, we just return the front portal fragments.

```
            // If the back is solid, just return the front list
            if ( CurrentNode->Back == BSP_SOLID_LEAF ) return FrontPortalList;
```

Now that we have a list of all the possible fragments that have survived the front tree (or the fragment that ended up in an immediate front list) stored in FrontPortalList, we will now loop through this linked list and send each portal fragment down the back tree one at a time (where they may also get split into multiple fragments). We collect the resulting fragments returned from clipping each front fragment to the back tree in a final linked list (called PortalList) which is returned from the function. This linked list will contain all the fragments of the original portal passed into the function that survived both the front and back trees of the current node.

The first thing we do is use the pPointer local variable to point at the head of the linked list. We then set up a while loop that will iterate this pointer though each portal in the list. The loop exits when we reach the end of the list and pPortal equals NULL.

```
            // Loop through each front list fragment and send it down the back
            pPortal = FrontPortalList;
```

```
            while ( pPortal != NULL )
            {
               CPortal * NextPortal = pPortal->NextPortal;
               BackPortalList       = NULL;
```

Each time through the loop, we store the next portal in FrontPortalList in NextPortal because the current portal will be getting sent down the back tree and possibly deleted. We certainly do not want to lose a pointer to the following portal in the list which was only linked to (and accessible via) the portal that got deleted. We send the current fragment (pointed to by pPortal) down the back tree, and store a pointer to the list of fragments that have survived in BackPortalList. BackPortalList will be NULL if no fragments of the current front list fragment being processed survive the back tree.

If some portals do survive, then we get the last fragment in the list and attach it to our master PortalList. Remember that BackPortalList will contain fragments that have survived for the current fragment that was in the front list. Any portals returned in BackPortalList are added to PortalList at the end of each loop. This means that when the loop exits, PortalList will contain a linked list of all fragments that survived both the front and back trees for each loop.

The next thing we do inside the code is test to see if the back child is a leaf or a node. If it is a leaf then the current fragment from the front list (pPortal) has made it into a back leaf, so we determine whether the leaf is in front or behind the portal and store the leaf index in the correct location in the portal's leaf array. If ClassifyLeaf returns NULL, then this is a rogue fragment and is deleted. If the portal does make it into a leaf, then we assign its address to BackPortalList since this will be the pointer that is linked to the master portal list at the end of the loop. If another node (not a leaf) exists down the back child, then we clip the fragment to the back tree and store any surviving fragments in BackPortalList . This is all shown in the following code snippet.

```
               // Empty leaf behind?
               if ( CurrentNode->Back < 0 )
               {
                   // The back is a Leaf, determine which side of the node it fell
                   LeafIndex   = abs(CurrentNode->Back + 1);
                   OwnerPos    = ClassifyLeaf( LeafIndex, pPortal->OwnerNode );

                   // Found the leaf below?
                   if ( OwnerPos != NO_OWNER )
                   {
                       // Attach it to the back list
                       pPortal->LeafOwner[OwnerPos] = LeafIndex;
                       pPortal->NextPortal          = BackPortalList;
                       BackPortalList               = pPortal;
                       pPortal->LeafCount++;

                   } // End if leaf found
                   else
                   {
                       // Delete the portal, but continue to the next fragment
                       delete pPortal;
                       continue;

                   } // End If no leaf found
```

```
                } // End if child leaf
                else
                {
                    // Send the Portal Down the back
                    BackPortalList  = ClipPortal(CurrentNode->Back, pPortal);

                } // End If child node
```

Remembering that we are still in the while loop for the current front list fragment we are processing, if BackPortalList does not equal NULL, then it means that we have a fragment (or potentially a list of fragments) that have survived both the node's front and back trees. If this is the case, we use a temporary portal pointer (Iterator) to seek to the end of the linked list (pointed to by BackPortalList). Once the end of the portal list has been reached, we assign the last portal in the list to point to the first portal in the master portal list (PortalList) so that any fragments we have collected in previous iterations (in Portal List) and the fragments we have just collected (in BackPortalList) are joined into a single list. PortalList is then reassigned to BackPortalList so that it points at the head of this merged list.

```
                // Anything in the back list?
                if (BackPortalList != NULL)
                {
                    // Iterate to the end to get the last item in the back list
                    Iterator = BackPortalList;
                    while ( Iterator->NextPortal != NULL)
                            Iterator = Iterator->NextPortal;

                    // Attach the last fragment to the first fragment
                    // from the previous iteration.
                    Iterator->NextPortal = PortalList;

                   // Portal List now points at the current complete list of
                   // fragments collected so far
                   PortalList = BackPortalList;

                  } // End if BackPortalList is not empty
```

At this point, the previous contents in PortalList and the contents in BackPortalList (generated from the current fragment being processed) have been joined into a single list, and PortalList points to the head.

With the current front list fragment processed and any fragments of it that survived the back tree added to PortalList, we now move on to the next fragment in the list. We do this by assigning pPortal to NextPortal. Remember that NextPortal was assigned to point at the next portal that needed to be processed at the head of this list.

```
                  // Move on to next portal
                  pPortal = NextPortal;

             } // End While

            // Return the full list
```

```
            return PortalList;
```

Finally, after the while loop exits, PortalList will contain any fragments that survived both the front and back trees of the current node, so we return it. That is the (rather tricky) on plane case taken care of.

CP_SPANNING

The final case is the spanning case which occurs when the passed portal is spanning the current node being visited. When this is the case, the portal is split into two child portals. Each child portal is clipped to the front and back tree respectively and the resulting fragments retuned from each sub-tree are linked into a single list and returned.

The first thing the function does is allocate the two new child portals that will be the by-product of the split. The current plane and these portal pointers are then passed into the current portal's Split function. This is just a standard polygon splitting function which we have seen before. When the function returns, FrontSplit and BackSplit will contain the two child portals and the original portal can be deleted.

```
      case CP_SPANNING:

            // Allocate new front fragment
            FrontSplit = new CPortal;
            BackSplit  = new CPortal;

            // Portal fragment is spanning the plane, so it must be split
            if (FAILED( pPortal->Split(*CurrentPlane, FrontSplit, BackSplit)))
            {
                delete FrontSplit; delete BackSplit;
                throw BCERR_OUTOFMEMORY;
            } // End If

            // Delete the ORIGINAL portal fragment
            delete pPortal;
            pPortal = NULL;
```

Now it is time to send each of these child portals down their respective sides of the tree, starting with the front split. If the current node has a leaf immediately in front of it, then we classify the leaf and store its index in the portal. We also assign the FrontPortalList pointer to point at this portal instead of just returning (we will see this being used in a moment). If NO_OWNER is returned from the leaf search, then this is a rogue fragments and is deleted. Alternatively, if a front child node exists down the front tree, we clip the front split portal to the front tree and store the surviving fragments in the FrontPortalList pointer.

```
            // There is another Front NODE ?
            if (CurrentNode->Front < 0 )
            {
                // The front is a Leaf, determine which side of the node it fell
                LeafIndex   = abs(CurrentNode->Front + 1);
                OwnerPos    = ClassifyLeaf(LeafIndex, FrontSplit->OwnerNode );

                // Found the leaf?
```

```
                if ( OwnerPos != NO_OWNER)
                {
                    FrontSplit->LeafOwner[OwnerPos] = LeafIndex;
                    FrontSplit->NextPortal          = NULL;
                    FrontPortalList                 = FrontSplit;
                    FrontSplit->LeafCount++;

                } // End if leaf found
                else
                {
                    delete FrontSplit;

                } // End If no leaf found

            } // End if child leaf
            else
            {
               FrontPortalList = ClipPortal(CurrentNode->Front, FrontSplit);
            } // End If child node
```

At this point, FrontPortalList will either point to the portal that made it into the immediate front child leaf, the list of fragments that survived when the front split was clipped to the front tree of the current node, or NULL if no fragments survived the front tree clipping process. Either way, we have a list that describes what fragments of the front split portal (if any) survived.

Now we will do the same for the back split. If the current node has solid space behind it then the back split is immediately deleted. If the node has an actual leaf behind it, the leaf is classified against the portal and its index is stored in the portal's leaf array. If the NO_OWNER result was returned from the leaf classification, then the portal is a rogue and can be deleted. If however a child node exists behind the current node, we will clip the back split portal to the back tree and store a pointer to the linked list of any surviving fragments in the BackPortalList local variable.

```
            // There is another back NODE ?
            if ( CurrentNode->Back == BSP_SOLID_LEAF )
            {
                // We ended up in solid space
                delete BackSplit;

            } // End if solid leaf
            else if (CurrentNode->Back < 0 )
            {
                // The back is a Leaf, determine which side of the node it fell
                LeafIndex   = abs(CurrentNode->Back + 1);
                OwnerPos    = ClassifyLeaf(LeafIndex, BackSplit->OwnerNode );

                // Found the leaf?
                if ( OwnerPos != NO_OWNER)
                {
                    BackSplit->LeafOwner[OwnerPos] = LeafIndex;
                    BackSplit->NextPortal          = NULL;
                    BackPortalList                 = BackSplit;
                    BackSplit->LeafCount++;
```

```
                } // End if leaf found
                else
                {
                    delete BackSplit;

                } // End If no leaf found

            } // End if child leaf

            else
            {
                BackPortalList = ClipPortal(CurrentNode->Back, BackSplit);

            } // End If child node
```

At this point we potentially have two fragment lists containing portals fragments that survived the front *or* back of the tree. We now need to merge FrontPortalList and BackPortalList into a single list so that we can return a pointer to this new merged fragment list.

In the first step, if FrontPortalList does not equal NULL, we iterate to the end of the list so that Iterator points to the final portal in the list. We then set the NextPortal member of this final pointer in the front list to point at the first pointer in the back list. This will connect the two lists and we can then return the FrontPortalList (which now points to the start of the combined list).

```
            // Find the End of the front list and attach it to Back List
            if (FrontPortalList != NULL)
            {
                        // There is something in the front list
                        Iterator = FrontPortalList;
                        while (Iterator->NextPortal != NULL)
                              Iterator = Iterator->NextPortal;

                        if (BackPortalList != NULL)
                            Iterator->NextPortal = BackPortalList;

                return FrontPortalList;

            } // End if front list
```

If there are no fragments in FrontPortalList, then we return just BackPortalList. Of course, if BackPortalList equals NULL, none of the split portals survived and we return NULL.

```
            else
            {
               // There is nothing in the front list simply return the back list
                if (BackPortalList != NULL) return BackPortalList;
                return NULL;

            } // End if no front list

            // If we got here, we are fresh out of portals so simply return NULL.
             return NULL;
```

```
    } // End switch
    return NULL;
}
```

That concludes the ClipPortal function and finalizes our discussion of the portal clipping procedure. The generation of portals is perhaps the most complicated component of the PVS calculator to develop since it is highly recursive. In fact, when we call the ClassifyLeaf function, we are actually invoking a recursive tree traversal function from within a function that is already recursively walking the tree. Although this can be a bit tricky to follow at first, hopefully the underlying logic is clear to you.

17.5.5 Portal Generation Conclusion

You should not be too hard on yourself if some of this lesson has been extremely hard to digest. Although the portal generation process is really just another form of CSG, it does have certain particular rules which make the process appear a lot harder than it actually is. While the ClipPortal function itself might look quite intimidating at first glance, most of the code is obfuscated by linked list management code and other such housekeeping tasks. But notice that the basic idea is CSG through and through. In fact, the entire process of clipping the portal to a solid BSP tree and keeping any pieces that ended up in empty space is taken directly from the CSG material that we learned about in the last chapter.

As a reminder, the ClipPortal method was called from the ProcessPortals function for each node that had two children. Any fragments that were returned to ProcessPortals from ClipPortal were considered valid portals that lived in two leaves. These portals were then added to the PVS processor's main portal array. At the end of the process, this master array will hold every single portal that plugs every single gap between every single leaf in the level. Each portal will also describe which leaves it is contained in and their location with respect to the portal's plane (front or back). In short, we now have the doorways between leaves at our disposal (as a list of portal polygons) and thus we have everything we need to enter the next phase of calculating potential visibility.

17.6 Calculating the Potential Visibility Set

After the last section, we now have all of the information necessary to calculate the PVS. Recall that in order to determine what was visible from one leaf to the next we need to know the size of the doorway (the gap) that joins those two 'rooms'. With our portals now in place, we have exactly that. Each portal describes the precise doorway between one leaf and the next, so the dimensions of the portal provide the dimensions of the gap. The next part of this course is arguably the most exciting – writing our PVS Calculator. While it will be one of the most complex and challenging discussions to date, it will also prove to be one of the most beneficial to our eventual engine (finally coming to fruition in Module III) in terms of sheer performance enhancement.

At a high level, our first task moving forward will be to loop through each portal in our portal list and determine exactly which leaves that portal can 'see'. Each portal in our master portal list will have a leaf visibility list constructed for it. The compilation of the PVS is primarily going to be occupied with calculating the leaf visibility sets for each leaf. Finally, once we know which leaves can be seen by all of the portals, we can easily find out which leaves a particular leaf can see, because it will be the sum of what all of the portals in that leaf can see. For example, imagine that leaf 1 contains portals A, B, and C and that the leaf visibility sets calculated for each portal are given by the following table.

Portal	Visible Leaves
Portal A	1,2,7,101
Portal B	20,21,22
Portal C	30,31,32

It stands to reason that since all of these portals are essentially doorways out of leaf 1, leaf 1 can see everything that can be seen through all of its portals. Therefore, the visibility list for leaf 1 will be the sum of portal visibility sets for each of the portals it contains. With the portal visibility sets at its disposal, the PVS calculator can quickly deduce that the visibility set for leaf 1 would be:

Leaf	Visible Leaves
Leaf 1	1, 2, 7, 20, 21, 22, 30, 31, 32, 101

Thus, after the PVS calculator has calculated the visibility sets for each portal, it will loop through each leaf in the tree, find the portals that are contained in each leaf, and then sum their visibility sets to create the final visibility set for that leaf. Once the visibility set for every leaf has been calculated, the portals can be discarded and the master PVS data array (which contains the visibility information for every leaf in a continuous block) can be compressed and saved out to file with the rest of the leaf tree data.

Summing the portal visibility sets to create the leaf visibility sets is performed in the final stage of the PVS Calculator and it is quite a trivial task. It is the calculation of the visibility sets for each portal which will require the most work.

The steps involved in the PVS Calculator design are outlined below:

- **Step 1: Creation of one-way portals**
  Each portal in our portal array currently lives in two leaves. We will duplicate each portal so that the N portals in our list are converted into N*2 portals. Each portal's duplicate will have its normal negated so that it faces in the opposite direction. We will then assign each original portal to the leaf in its backspace. The leaf contained in each portal's frontspace will be known as the neighbor leaf. This will create a list of portals in which the portals are only owned by one leaf; the leaf from which the back face of the portal can be seen. Each portal will also store the index of the neighbor leaf. Note that this is the leaf that you would arrive in if you traveled from the owner leaf through the back of the portal it owns. The normal of each portal will point into the neighbor leaf and portal flow is assumed to only happen through the backs of portals. That is, if you cannot see the back of a portal, you cannot see through it (more on this later). As an example, imagine we have an original portal P that is contained in leaves 5 and 6, with leaf 5 in its frontspace. Portal P will have its index stored in leaf 6 since this is the leaf that owned the backspace of the portal. Leaf 5 will be stored in the portal as the neighbor leaf (i.e., the leaf we travel into when passing from leaf 6 through the back of the portal). Once this is done, we will then duplicate the portal, reverse the normal, and swap the owner/neighbor relationship in the new duplicate.

  If we imagine that the original portal P contains leaves 5 and 6 in its leaf list, with leaf 5 in its front halfspace, two new portals would be created to replace the original portal and they would have neighbor/owner properties as shown in the following table.

Portal	Front Space	Back Space	Owner	Neighbor
Portal P1	Leaf 5	Leaf 6	Leaf 6	Leaf 5
Portal P2	Leaf 6	Leaf 5	Leaf 5	Leaf 6

  While this may seem like an odd thing to do, it actually simplifies the main PVS processing procedure and speeds up compilation time for the PVS data. Not having these optimizations could mean the difference between the PVS calculator taking 10 minutes to compile versus several hours (if not more).

  We will see why this is so important in a moment. For now, just notice in the table above that a leaf only ever owns portals whose normal face outwards from that leaf. This allows us to establish a portal "flow direction" such that we can assume that we can only see through the back of a portal and never see (or step) through the front of one. This allows us to avoid having to run tests while in the core of the recursive loop to make sure that we do not step into leaves we have already visited or step through portals which face in the opposite direction to which the visibility flow is currently taking us. Essentially, because a leaf only contains the portals that face out of it, we know that we are only ever supposed to see through the backs of portals. Portals that are facing us should be ignored. This prevents us from traversing through portals for which no line of sight could possibly exist between the portal currently having its PVS generated and the target portal. We will discuss this in more detail shortly.

- **Step 2: Portal To Portal Visibility**
  Because the core PVS calculator is basically a highly recursive clipping procedure, it can take a very long time to calculate the visibility information for each portal. The basic procedure is to

build something called an anti-penumbra from a series of clip planes so that we can model the way light would travel out from a portal into the neighboring leaf. The source portal is the portal that is currently having its leaf visibility set calculated and as such is used to construct a viewing volume (a loose analogy that will be more accurately explained shortly). Any portals in neighboring leaves that fall within the anti-penumbra of the source portal are considered visible from the source portal. As such, we recur through that portal and into its neighboring leaf. Once in that leaf, it is marked as visible and its portals are tested for anti-penumbra containment. Any portals in that leaf that fall within the anti-penumbra of the source portal are once again stepped through, taking the recursive process into its neighboring leaf, and so on. Eventually, we will step into a leaf where none of its portals will fall within the source portal's anti-penumbra. In this case, the recursive process ends along that path for the source portal. We have essentially reached a leaf in which the source portal cannot see any of its other portals. Therefore, it cannot see *through* those portals into other leaves.

When we consider that this process will be repeated for each portal, there is going to be an incredible amount of processing to do. As such, we will need to include some small optimization steps before the core process begins to reduce the amount of leaves/portals that would be unnecessarily visited during the key recursive clipping process. The first thing we will do is build a temporary portal visibility list for each portal. That is, each portal will store an array describing which other portals can possibly see it (a very rough approximation). The fact that we have created one way portals makes this process very easy as we will discuss in moment. For example, imagine that we wish to calculate a very rough portal visibility set for portal A. We can loop through all the portals and reject portals from its visibility set based on two key things. If the portal being tested has its normal facing portal A then these one-way portals are facing in opposite directions and portal A cannot possibly see through the back of test portal B (i.e., it cannot see through it). Such portals will not be added to portal A's rough portal visibility list. Secondly, if the test portal is behind portal A then once again, portal A cannot possibly see through it. Remember, our one-way portals have normals that face into the neighbor leaf and as such, visibility flow can only travel through the back of a portal into the leaf it leads into. If portal B is behind portal A, portal A cannot look through the back of portal B, so no visibility exists between them.

At the end of this step, for each portal we will have generated an array of bytes equal to the number of one-way portals in the scene. Each element in each portals array will describe the visibility of the corresponding portal in the master portal list. For example, if portal 5 has the 25^{th} byte in this array set to 1, then it means that portal 5 was found to have some chance of possibly seeing through portal 25. As such, this relationship will have to be examined more closely by the core PVS calculator. Although this is an extremely rough approximation, it does tell us what portal combinations cannot possibly see each other and thus should not be processed during the core recursive process. We will discuss this in more detail later.

- **Portal Leaf Flood Fill**
  After creating the portal visibility array for each portal we will use it to perform a very fast flood fill through the leaf to generate a very rough portal/leaf visibility array. At this stage, no anti-penumbras will be generated and clipped and we will be creating a very rough leaf visibility set

for each portal. It is this set that will be further refined by the core PVS processor to get the final visibility sets for each portal.

In this step a very small recursive function will be invoked for each portal. Starting at the portal neighbor leaf if will flow though all its portals into their neighbor leaves and so on. Every time a leaf is reached by the recursive function, its bit is set in the portal's visibility set. Now, you may be thinking that such a procedure would always recursively flood through all portals until every leaf in the level has been added to the visibility set for each portal. Indeed if it were not for the portal to portal visibility set that we calculated in the previous section, this would be the case. However, you will see in a moment that because we already know what portals a given portal can (roughly) see, the flood fill stops and returns as soon as it reaches a portal that is not set as visible in the source portal's 'portal visibility' array. Therefore, the portal/portal visibility array we process in the previous step is used as input to this step to perform a quick and approximate flood fill through the level, adding any leaves it reaches to the portal's leaf visibility set. This will be a *very rough* PVS and will contain many leaves that cannot actually be seen from each portal. However, it will also reject a lot of leaves and these are leaves that we will not have to step into during the core recursive process.

These last two steps serve no purpose other than to speed up the core PVS calculator. If you have ever had the misfortune of running a PVS calculator that did not employ these optimizations, you will know how vital they are. Complex levels could take many hours (perhaps even days) to compute, but this can be cut down to a mere fraction of the time with the portal/leaf flood fill in place as a preliminary measure.

- **Recursive Anti Penumbra Clipping**
  This final phase is the core PVS calculator that recursively floods through the level building the actual PVS from a combination of the preliminary PVS (built in the portal flood phase in the previous step) and from a good deal of clipping of a visibility volume. This visibility volume is called an anti-penumbra and essentially models everything that can be seen through a combination of two portals. The first portal is always the **source portal** (i.e., the portal for which the leaf visibility set is currently being compiled). The second portal is referred to as the **target portal** and it changes each time we step into a new leaf.

Before we discuss the core process that performs the recursive anti-penumbra clipping to calculate the actual visibility set, we will first talk about what an anti-penumbra is, how we create one, and how it is used to calculate the leaf visibility for each portal.

17.6.1 Anti-Penumbras

In discussions of illumination, an *umbra* describes a volumetric region that is completely in shadow (i.e. the darkest part of a shadow). A *penumbra* describes a partially shadowed volumetric region that exists between light and shadow. An *anti-umbra* is the opposite of an umbra and describes a region of total illumination. Thus, an *anti-penumbra* is a region of partial illumination. The real difference here is with respect to the occlusion or non-occlusion of light. In the case of an umbra and penumbra, light from a

light source is blocked by an object and some region on the opposite side of that object is shadowed. We can think of this as ‘shadow casting’. With anti-umbras and anti-penumbras, we can think instead of the light as passing through a gap and shining into an area.

Figure 17.37 depicts the concepts discussed. In the top diagram we see an occluder that creates areas of shadow opposite the light source. These areas are marked as umbra and penumbra to describe the shadow characteristics. The bottom diagram demonstrates the concept of light flowing through the gap between two occluders. The light creates an anti-umbra and an anti-penumbra as described above.

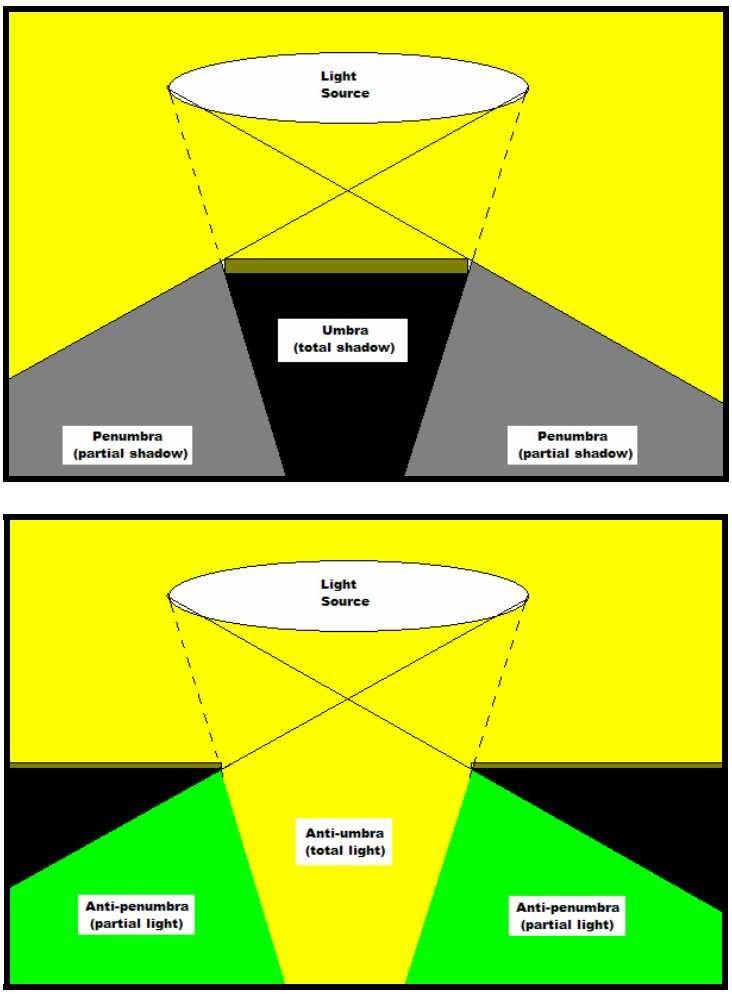


Figure 17.37

For our purposes, the anti-penumbra encompasses the *entire region* where light travels, including the anti-umbra. Thus, by calculating the geometric volume for the anti-penumbra, we are calculating the entire volumetric space where light travelling through a gap can go. Any objects in the anti-penumbra will receive some amount of light. Obviously those directly in line with the light source would be more illuminated, but for our purposes, this is not important. What is important is that from the perspective of the light source, the anti-umbra plus the anti-penumbra represent the total area that can be ‘seen’ (if we take the standard definition that in order for us to ‘see’ something, light must reflect off of it).

So we see that our situation is akin to a doorway between rooms. If one room (A) contains a bright light source and the other (B) does not, so long as the door is closed, B remains dark while A is lit. As soon as the door is opened, light from room A flows into room B and some portion of B is now illuminated. How much of room B is affected depends entirely on the size and shape of the doorway between them, since this is the only means by which light can enter room B.

Some parts of room B likely remain in shadow because they are positioned at such an angle to the doorway that they still remain in the region just outside of the anti-penumbra. So our anti-penumbra describes everything that can be seen in room B through the doorway when standing anywhere in room A. Any areas of B that remain in darkness cannot be seen from room A no matter where one stands in room A because no light has reached them.

Hopefully the applicability of an anti-penumbra to our portals (doorways) between leaves (rooms) is becoming more obvious to you.

17.6.2 Creating Anti-Penumbras

In our PVS system, an anti-penumbra will be constructed using some number of clip planes as we enter each leaf in the recursive process and it will extend between two portals: a source portal and a target portal. Do not worry for now about what portals should be used for the source and target portal; the important point in this section is the realization that building an anti-penumbra from a series of clip planes between the source and target portals will tell us about everything that can be seen from the source portal through the target portal. This is the key idea behind calculating PVS. Before we see this in action, let us first talk about how we build an anti-penumbra between any two portals.

The anti-penumbra is a collection of clipping planes, not unlike those used by D3D to reject geometry outside the viewing frustum. Therefore, although we have used the analogy in an earlier discussion that calculating the PVS is essentially like filling a leaf with light and recording the visible leaves as the ones that get touched by that light, no lighting formulas occur in this process whatsoever. We are simply constructing clip planes to represent the area that the anti-penumbra will touch. Any portals/leaves not inside this collection of clip planes will be considered to be invisible to the source portal.

As we know, to create a plane in 3D space, we require only three points. Our job will be to loop through each edge in the source portal, and for each one of these edges, loop through each vertex in the target portal. With the two vertices from our source edge and the vertex from the target portal we now have access to three points, which allow us to construct a plane. The pseudo-code looks something like the following:

- For Each Vertex In Source Portal (Source Vertex 1)
    - For Each Edge in Target Portal (Target Vertex 1, Target Vertex 2)
        - Create Plane using Source Vertex 1, Target Vertex 1, and Target Vertex 2
        - Is Plane an Anti-Penumbra Clip Plane?
            - Yes -- Add to Clip Plane List
            - No -- Ignore it and move on to next Edge
    - Next Edge in Target Portal
- Next Vertex in Source Portal

This approach obviously generates many planes but only those that are considered to be anti-penumbra planes are added to the clip list -- the rest are ignored. So the important question is “what is an anti-penumbra plane?”

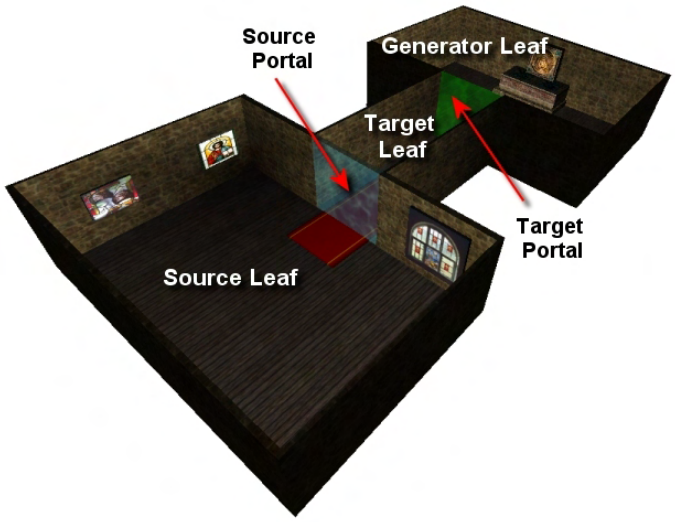


Figure 17.38

A plane will be considered an anti-penumbra plane if it clearly divides the source portal and the target portal. That is, the source portal must be totally on one side of the plane and the target portal must be totally on the other. To show an example of generating an anti-penumbra plane, we will use the simple three leaf construct shown in Figure 17.38. The portal having its leaf visibility information calculated is labeled ‘Source Portal’ in the image. This portal is assumed to be owned by ‘Source Leaf’ and face into the leaf ‘Target Leaf’. The other portal in the target leaf is referred to as the target portal as this is the portal that leads out of the target leaf into the third leaf in the run, the generator leaf. Do not worry too much about why the generator leaf is called what it is at the moment. Just know that for any portal that is current having its leaf visibility set calculated, the source leaf, initial target leaf and the generator leaf will always be visible. This makes sense as the source portal leads directly into the target portal and as such, if you are were located inside the source portal you would always be able to see though this doorway into the target leaf. Therefore, the ***initial*** target leaf of the source portal is always visible.

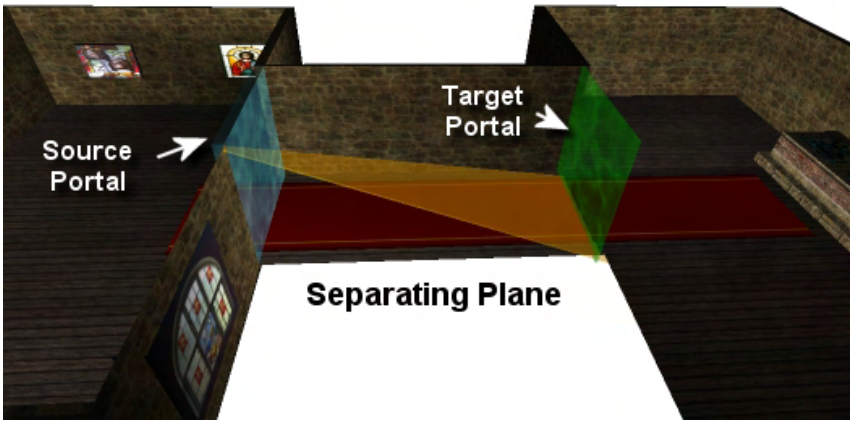


Figure 17.39

Note that we used the word ‘initial’ in the previous sentence as the target and generator portals change throughout the recursive process as we visit different leaves. Similarly, the initial target portal is also always considered visible to the source portal. This is intuitive when we consider that the source portal leads into the target leaf and therefore must have some visibility with respect to the portals leading out of that leaf. Although the target portal will change as we step through each leaf during the PVS calculation process, the source portal will always remain constant as it is the portal currently having its leaf visibility calculated.

The source and target portals are always used to construct the anti-penumbra planes at each step. As mentioned, the construction of the anti-penumbra involves finding all planes that separate the source and

target portals into two separate halfspaces. This is done primarily by looping through each vertex in the source portal and each edge in the target portal and creating a plane from these three points. The two portals are then classified against this plane and considered to be an anti-penumbra plane only if the classifications of each portal are different. In Figure 17.39 we can clearly see a separating plane being created from the top left vertex of the source portal and the bottom edge of the target portal. If we consider the plane normal to be facing upwards, the source portal is clearly in the backspace of the plane and the target portal is in its front space. This is a valid anti-penumbra plane.

In Figure 17.40 we see another plane being constructed from the same vertex in the source portal and the top edge of the target portal. Assuming the normal of this plane is pointing upwards in this image, we can see that both portals exist in the backspace of the plane and therefore this is not a separating plane. This plane would be rejected from the anti-penumbra generation process as it does not describe any extreme visibility angle between the source and target portal.

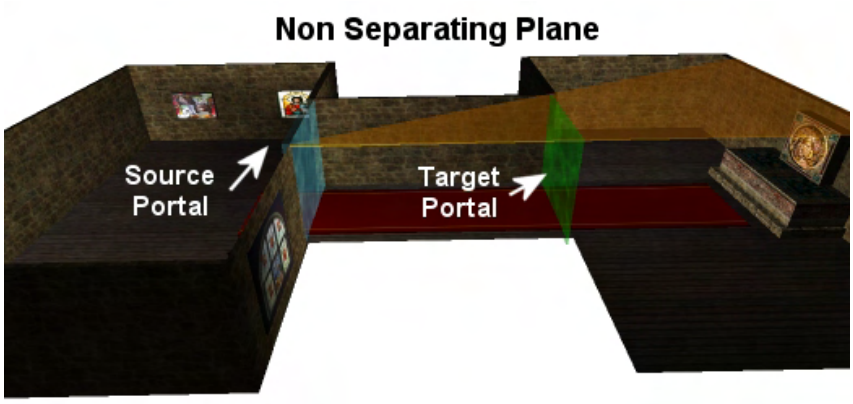


Figure 17.40

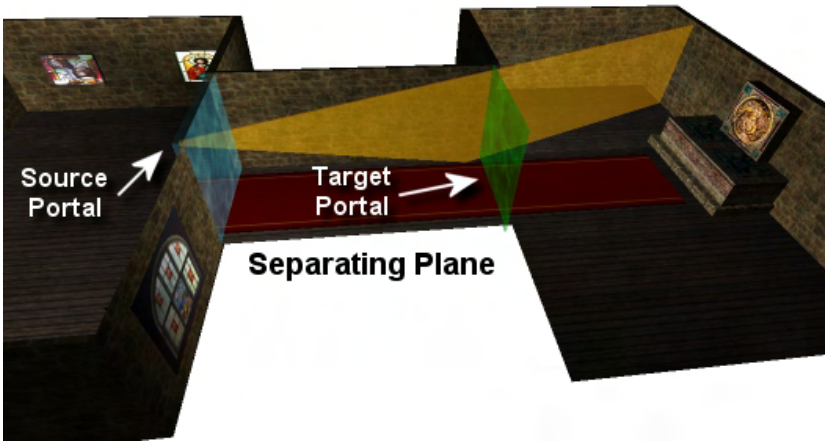


Figure 17.41

In Figure 17.41 we test another plane using the same vertex in the source portal and the far edge of the target portal. Assuming the normal is facing away from us in this example, we can see that the source portal would be contained in the plane's frontspace and the target portal in its backspace. This clearly makes it a separating plane as it describes the extreme viewing angle that can be experienced from the near side of the source portal though the far side of the target portal. This plane would be considered a valid anti-penumbra plane and would be added to the list of clip planes that will be used later for the rejection of portals in the generator leaf.

Finally, we show one more example of a separating plane so that we are in no doubt as to exactly how the anti-penumbra planes should be constructed. In Figure 17.42 this time we use the near bottom vertex of the source portal and the two vertices that form the far edge of the target portal. Assuming the normal is facing away from us in this example, the source portal would be contained in the plane's frontspace and the target portal is contained in the plane's backspace.

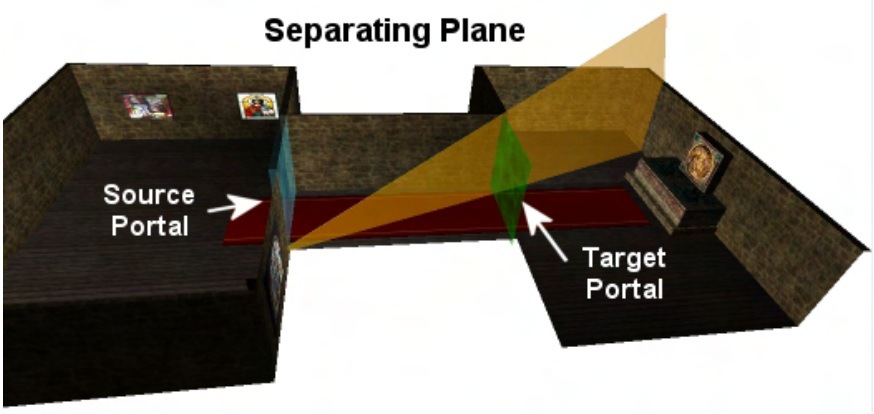


Figure 17.42

When building this view volume from the source portal to the target portal we create planes from every vertex in the source portal to every edge in the destination portal. The result is many accepted planes which describe the extremities of what can be viewed from the source portal through the target portal and into what is called the *generator leaf*. The generator leaf is always the leaf which the target portal leads into. It is the leaf in which the anti-penumbra planes form the volume for which everything inside is considered to be visible from the source portal. To get the general idea across more clearly we see a simplified anti-penumbra constructed using only two planes in Figure 17.43.

In this example we can see that the formation of the planes as they enter the generator leaf describe everything that can be seen from any position in the source leaf. The target leaf is always visible to the source portal as this is the leaf the portal leads into.

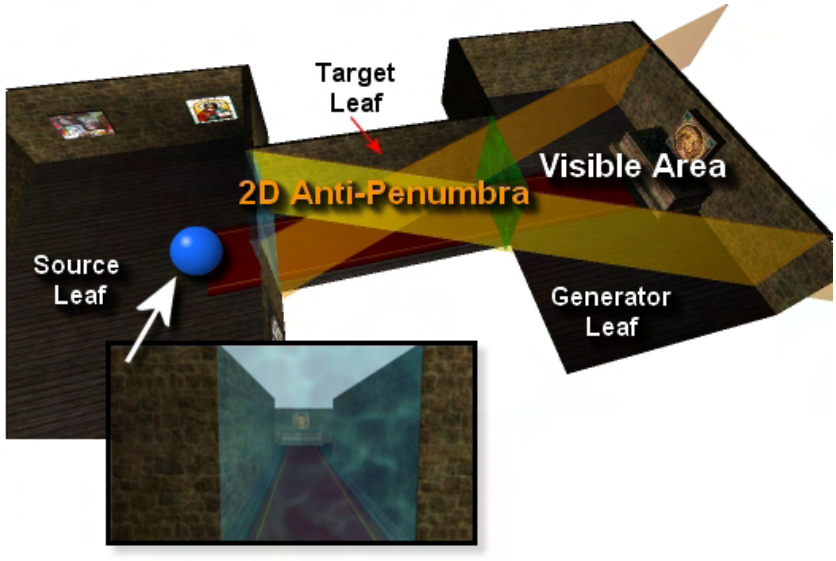


Figure 17.43

In Figure 17.43 the blue sphere represents a potential viewing position where the camera may be located at run time. The square inset in the image shows what can be seen from this point of view. As you can imagine, if the sphere was moved slightly to the left or right it would increase what can be seen in the generator leaf to the left and right of the target portal, respectively. The formation of the anti-penumbra planes in the generator leaf completely and conservatively bound this region so that any area inside those planes is always considered visible. Although we will examine the recursive process in a moment, it is worth noting that if any portals exist in the generator leaf, we recur through the portal into that leaf. No portals are in the generator leaf in this example, but if there were and they were contained inside the anti-penumbra, it means that this doorway is visible (potentially) from the source portal and we should step through that portal and into its neighbour leaf. That neighbour leaf is then considered visible from the

source portal. If a generator portal is outside the anti-penumbra in the generator leaf, we do not have to step through that portal and into its neighbour leaf. It is intuitive that if the doorway into the leaf cannot possibly be viewed from the source portal, then the leaf on the other side of that doorway cannot possibly be visible either.

If portals are located in the generator leaf we refer to them as *generator portals*. This is because, if they do fall within the anti-penumbra, another recursive step is 'generated' as we step through that portal and into its neighbour leaf (marking the neighbour leaf as visible). When we recur into the neighbour leaf of a generator portal, the leaf on the other side becomes the new target leaf and what was previously the visible generator portal becomes the new target portal and the anti-penumbra is constructed all over again between the source portal and the new target portal. The portal in this new leaf then becomes the new potential generator portal and the process continues until we step into a leaf where no generator portals exist inside the anti-penumbra. At this point we have exhausted the visibility of the source portal along that path of portals. We will look at some examples of this entire recursive clipping process in a moment.

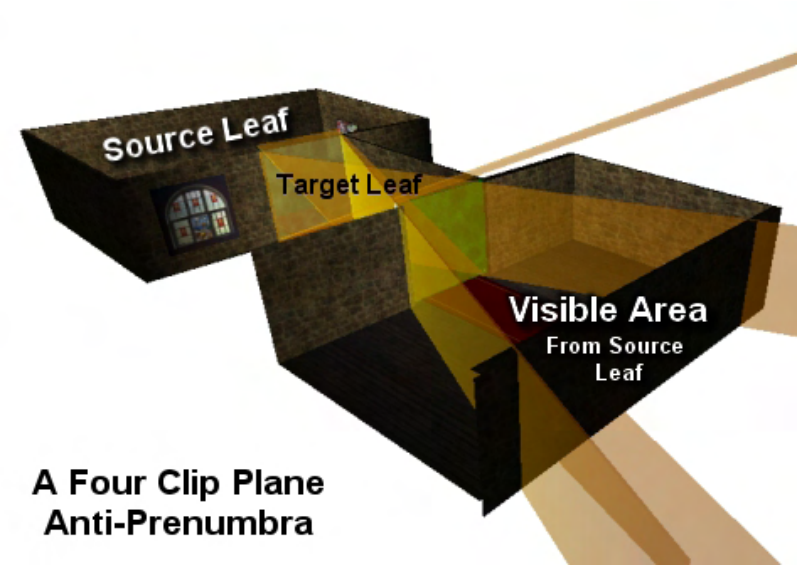


Figure 17.44

To give you a more three dimensional idea of what the anti-penumbra looks like in the generator leaf, we see a four plane example in Figure 17.44. Of course, the real anti-penumbra would have many more planes than this, but it gives us a good idea about the viewing volume created in the generator leaf. Any portals in the generator leaf that lay behind all of these planes are inside the anti-penumbra and should be stepped into.

The reason why we generate planes from every vertex in the source portal to every edge in the target portal is that, while it may generate many duplicate (or nearly duplicate) planes, it makes sure that we get the tightest clip planes possible from that combination. However, there is more we can do to generate more clip planes that may produce some even tighter clip planes. This will allow us to shave off a little excess visibility that may be represented in the generator leaf by the anti-penumbra clip plane.

To get the tightest planes possible for our anti-penumbra, after we have tested every combination of vertices in the source portal with edges in the target portal looking for candidate planes, we can then search for clip planes the other way around as well. That is, we can test every vertex in the target portal against every edge in the source portal. In the above example this would generate the exact same planes and would therefore seem a little pointless. However, there is nothing that guarantees that the portals will be quads (they most likely will not be). Furthermore, the source and target portals may not even be the same shape. Therefore, if we also do the plane collection process in the reverse direction as well (target to source), we will end up possibly generating some unique planes that were not generated the

other way round. These planes may provide a slightly tighter clip. As Figure 17.45 shows, if two portals are different shapes, a completely different set of planes can be generated by reversing the order.

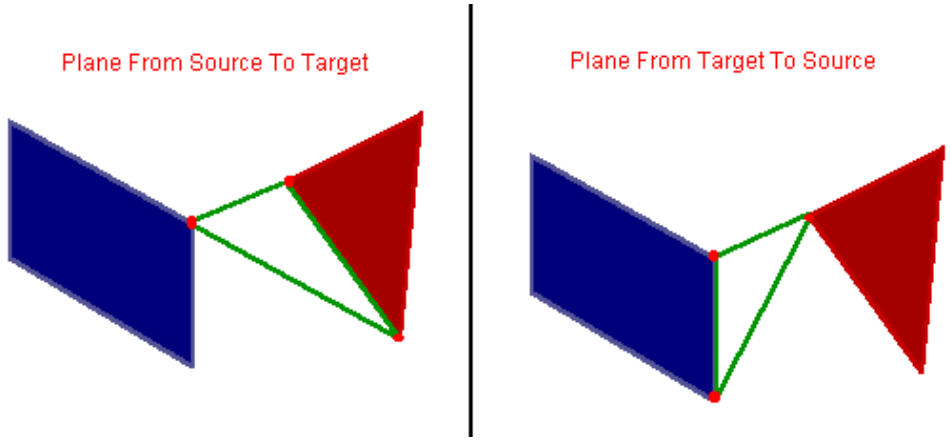


Figure 17.45

We can see in this example that creating a plane from the top left vertex of the quad and the bottom edge of the triangle portal is very different from the plane created using the far vertex in the target portal (the triangle) to the near edge of the source portal.

We now have a very good understanding of what the anti-penumbra is and how it will be constructed. Ultimately, it represents a visibility volume which describes what can be seen from the source portal through the target portal.

17.6.3 How Anti-Penumbras Determine Portal Visibility

Taking all of the previous optimizations out of the equation for the moment, the process of calculating the PVS is one of walking through the leaves of the tree and marking any leaves that we recur into as visible from the current source portal. Every portal is given its turn as the source portal when the time comes to calculate its visibility set. In this section we will step through a very small example that shows the recursive process involved in calculating the leaf visibility set for a single portal.

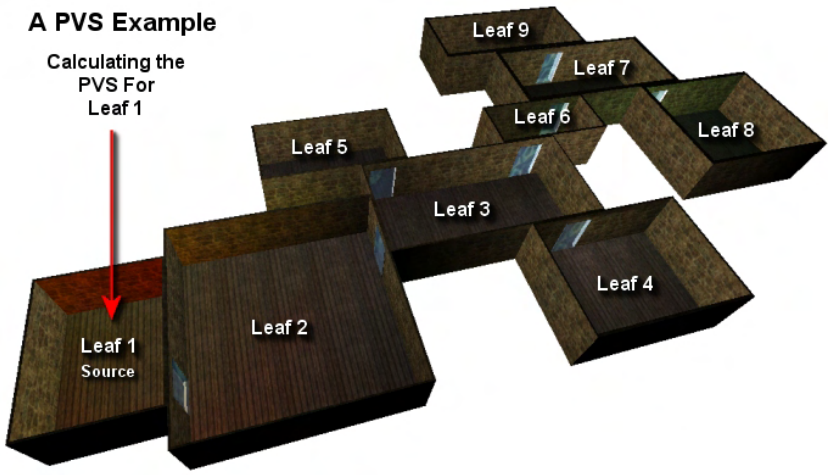


Figure 17.46

In Figure 17.46 we show some example geometry where each room is assumed to be a leaf for the purpose of explanation. The portals are shown as the blue doorways that connect the rooms. In our

following examples we are going to calculate the visibility set for the source portal. This is the only portal leading out of leaf 1 in the diagram. Incidentally, since this is the only portal in leaf 1, we also know that the leaf visibility set we calculate for this portal will also be the leaf visibility set for the source leaf.

When we first enter the recursive process, we know only the location of the source portal. Our first step is to fetch the index of the neighbor leaf from the source portal which will give us the target leaf (shown in Figure 17.47). In the target leaf we enter a loop that will build an anti-penumbra between the source portal and each target portal contained in that room. We then step through that target portal and determine which portals in the generator leaf are within the anti-penumbra and should be stepped through. In this example, there is only one portal in the target leaf, so this is the single target portal we must process. Having located the source and target portals we then construct the anti-penumbra between them as shown in Figure 17.47. These clip planes form a viewing volume in the generator leaf against which all portals in that leaf must now be tested.

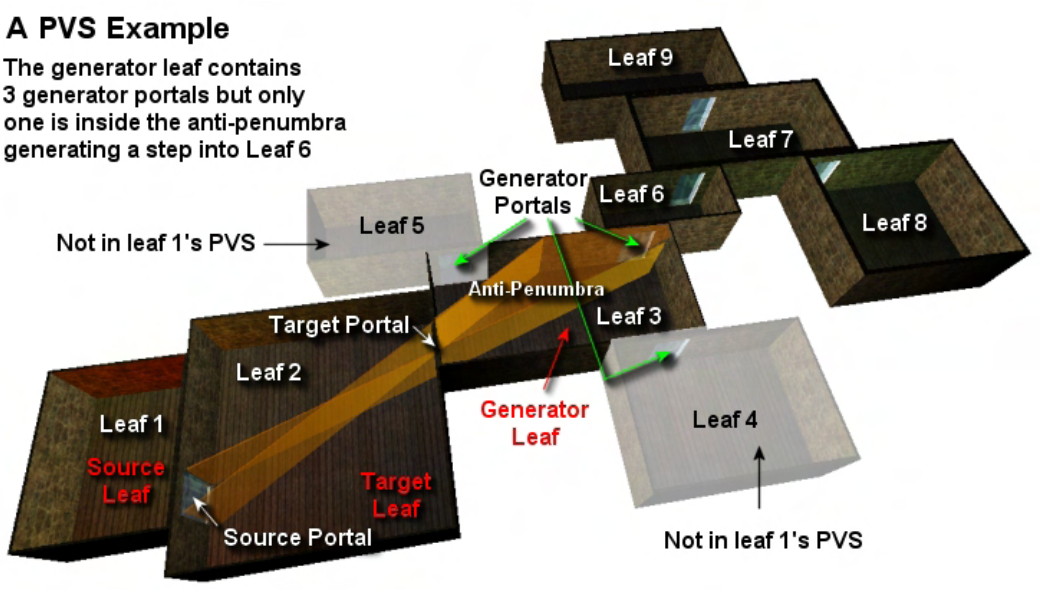


Figure 17.47

> **Note:** Any leaf we step into is instantly marked as visible in the source portal's visibility set. Thus, so far in our example, the target leaf and the generator leaf would have been added to the source portal's PVS.

We must now examine what happens in the generator leaf in Figure 17.47. The generator leaf is leaf 3 in the diagram and it has three portals (referred to as generator portals at this stage) which lead to leaves 4, 5, and 6. Each portal in the generator leaf is tested against the anti-penumbra clip planes and any portals that lay outside the anti-penumbra are considered invisible to the source portal and are ignored. This means that we will never step through these portals into leaves 5 and 6 and thus, we have just rejected leaves 5 and 6 from the source portal's (and therefore the source leaf's) visibility set.

Notice however, that the portal that leads to leaf 6 is contained inside the anti-penumbra. Therefore, we have to step through that portal. Since each portal has stored the index of its neighbor leaf, this will tell us that we need to step into leaf 6 next. When we step into leaf 6, the process repeats again, as shown in figure 17.48. However, when we enter the generator leaf, this becomes the new target leaf and the portal we just stepped through (the generator portal) becomes the new target portal via which a new anti-penumbra is constructed with the source portal.

As Figure 17.48 shows, once we have stepped through the generator portal into its neighbor (leaf 6), leaf 3 (the previous generator portal) becomes the new target leaf and any portals that previously passed the visibility test in the generator leaf become the new target portals. In this instance, where only one of leaf 3's portals was found to be visible in the previous recursion, the portal we stepped through into leaf 6 (i.e., the previous generator portal)

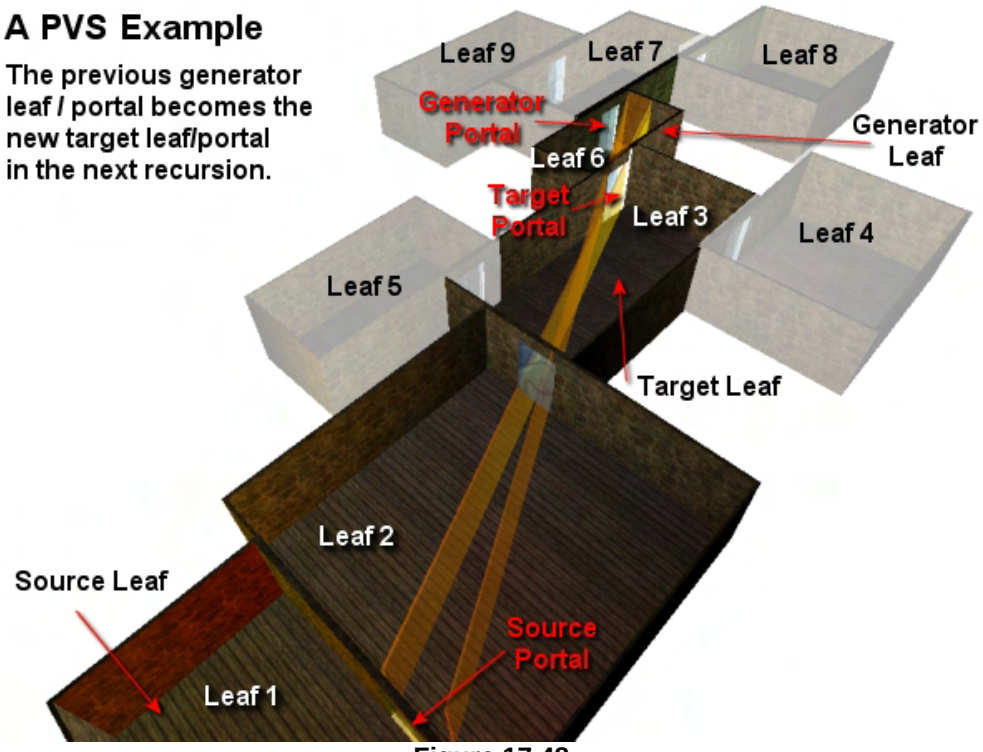


Figure 17.48

becomes the new target portal. In this second step, the anti-penumbra is constructed all over again between the source portal and the new target portal (which was the previous generator portal) which leads us from leaf 3 into leaf 6. Leaf 6, which we have just stepped into, has become the new generator leaf and, as we can see in Figure 17.48, the new source/target portal combination has generated an anti-penumbra that creates a view volume in this leaf. Our next step is to loop through each generator portal in the new generator leaf (leaf 6) and test them for inclusion in the anti-penumbra. Any portals that are contained or partially contained in the anti-penumbra are stepped through and become the new target portal in the next recursion. This same process continues until we step into a generator leaf where no portals are found to be inside the anti-penumbra and the recursion stops along that path. Remember, every visible portal in a given generator leaf has its chance at being a target portal, so you can imagine the high number of recursive iterations that can occur when determining everything that can be seen, even by a single portal.

As can be seen in Figure 17.48, as we stepped into leaf 6 and added it to the source portal's PVS, we found that none of its portals are contained inside the anti-penumbra. Therefore, we never recur into leaves 7, 8, and 9 and they are never added to the visibility set for the source portal. The final visibility set for the source portal in this example would contain leaves 1, 2, 3, and 6. This same process would be repeated for every portal in the level (i.e., every portal would have a chance at being the source portal for the calculation of its PVS).

So to summarize:

Because of the fact that BSP trees have convex leaves, we always know that the leaf on the opposite side of our source portal (i.e., the target leaf) is always visible. We then test each portal in the target leaf

(which we call target portals) and provided that the target portal is not on the same plane as the source portal, it too is always visible from the source portal.

> **Note:** If a target portal is on the same plane as the source portal, then we just ignore it.

If the target portal is visible, then it is obvious that the leaf on the other side of the target portal is also visible (Leaf 3 in our example). This leaf is called a *generator leaf* and it is where the recursive process begins for this current source/target leaf combination. Any portals in the generator leaf (called generator portals) that fall within the anti-penumbra are considered visible. Conversely, generator portals that are outside of it are ignored and not stepped through. Once we have the PVS for every portal in a given leaf, it is a simple matter to find the leaf's PVS. For example, the source leaf can see everything that the source portal can see. Therefore, *a leaf can see everything that all of its source portals can see.*

We now have some understanding of how a PVS calculator begins to calculate the PVS for a given portal. For every source portal in the source leaf, we create an anti-penumbra with every target portal in the target leaf. However, it is vitally important to make the anti-penumbra as small as we possibly can at each step as this allows us to refine our PVS set. If the anti-penumbra planes describe a view volume in the generator leaf that is unnecessarily large, it is possible that some generator portals will be stepped through that are not actually visible from the current source portal. This will make the source portal's (and therefore the source leaf's) PVS unnecessarily generous, which is not something we want.

We can refine the PVS by taking into account the fact that each generator portal that falls within the anti-penumbra at a given step in the process will become the target portal that is used to build the anti-penumbra in the following step. Therefore, if in the previous step only half the generator portal was contained inside the anti-penumbra, we can clip the generator portal (a copy of it) and build the anti-penumbra in the next step using this clipped version.

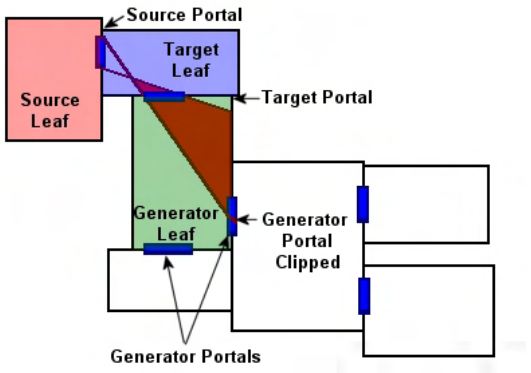


Figure 17.49

In the next step, the target portal (which was the clipped generator portal in the previous step) will be smaller, resulting in a smaller, less spread out anti-penumbra. This may allow us to reject portals in the next step that we would have otherwise had to have visited. To demonstrate this recursive generator portal procedure, we will look at an example from a top down view. The 2D nature of the following diagrams will help clarify what is actually happening.

We show the first step in the process in Figure 17.49. As discussed, the source, target and generator leaves are all marked as initially visible from the source portal in the first step. The clip planes (i.e., the anti-penumbra) are built from the source portal to the target portal. This describes how much of the generator leaf is visible through the target portal when viewed from the source portal. Of the two generator portals in this example, only one is partially within the clip planes. The portal outside anti-penumbra is ignored and will not be processed. The other portal is partially inside, so we clip away the part of the generator portal that is outside the anti-penumbra so we are left with just the fragment that is within the clipping volume. Because the generator portal is partially visible, it means that the leaf on the other side of this portal is also visible. We mark it as such by adding it to the source portal's PVS.

Next, this new leaf becomes the new generator leaf and the old generator leaf becomes the new target leaf. That is, the old (clipped) generator portal becomes the new target portal. So we now build a new anti-penumbra from the same source portal to the new target portal (the old generator portal that was clipped) which fully describes what volume in the new generator leaf is still visible. As we recursively move from leaf to leaf doing this (making the old generator portal the new target portal and rebuilding the anti-penumbra), clipping each generator portal to the anti-penumbra, the anti-penumbra gets smaller and smaller with each iteration until, in the end, no portals in the current generator leaf are inside. Because the generator portal is partially visible, it means that the leaf on the other side of this portal is also visible. We mark it as such by adding it to the source portal's PVS. The generator portal is also clipped, which leaves only a very small portal fragment (see Figure 17.50) surviving that will be used as the target portal in the next step.

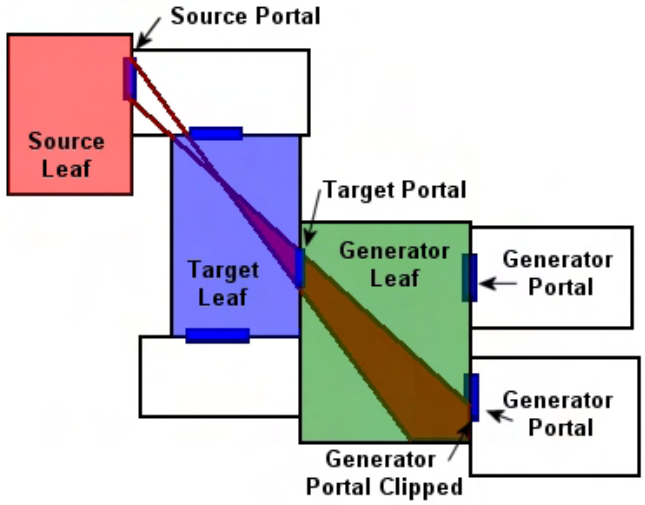


Figure 17.50

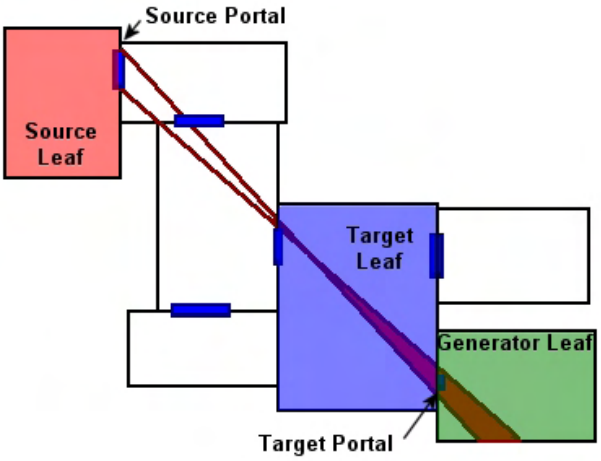


Figure 17.51

Next, this new leaf becomes the new generator leaf and the old generator leaf becomes the new target leaf. That is, the old generator portal becomes the new target portal. So we now build a new anti-penumbra from the same source portal to the new target portal (the old generator portal that was clipped) which fully describes what volume in the new generator leaf is still visible. As we recursively move from leaf to leaf doing this (i.e., making the old generator portal the new target portal and rebuilding the anti-penumbra) and clipping each generator portal to the anti-penumbra, the anti-penumbra gets smaller and smaller with each iteration until, in the end, no portals in the current generator leaf are inside.

17.6.4 Generator Portal Visibility

One of the last things we have to cover before looking at code is how to determine whether or not a generator portal is within the anti-penumbra volume. It turns out that this process is very easy. Figure 17.52 shows that the gray areas that are not visible are on the same side of the clip plane as the source portal.

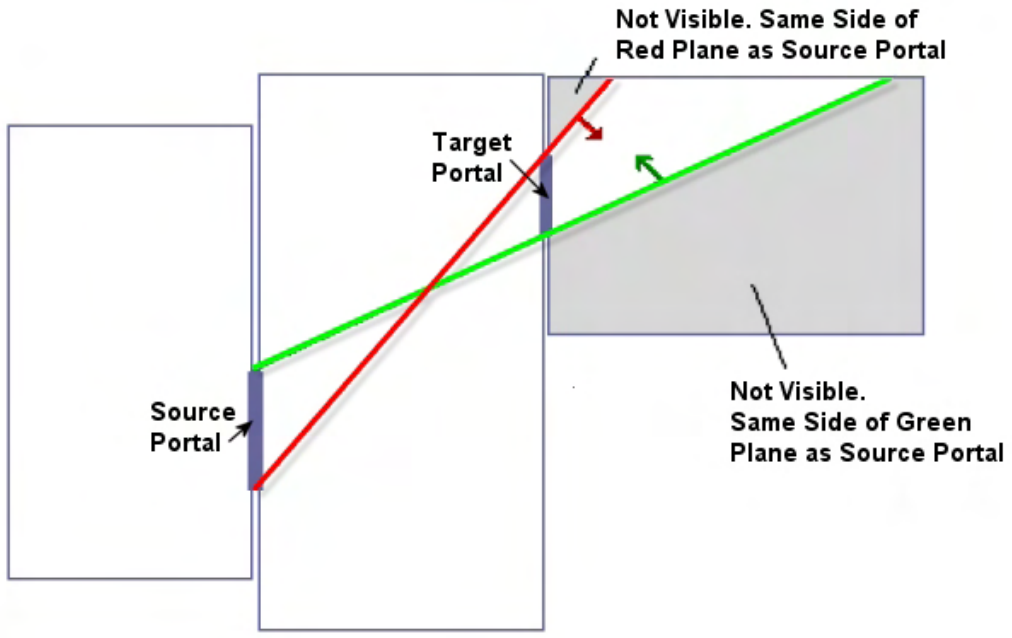


Figure 17.52

Our strategy is now pretty clear. We will check each generator portal against each clip plane of the anti-penumbra and if it is on the same side of the plane as the source portal, then it is not within the anti-penumbra and it can be rejected. This means that we will no longer need to recur into the leaf on the other side of that portal. Conversely, if a generator portal is on the opposite side of the plane with respect to the source portal, then it is within the anti-penumbra and should be tested against the other planes in the anti-penumbra. If a portal is spanning the plane, then it will be clipped to the plane, keeping the piece that is on the opposite side of the plane with respect to the source portal. Thus the entire PVS process is essentially one long recursive clipping process.

Unfortunately, while PVS is just a recursive clipping algorithm, our implementation will require some thought. Just doing a recursive clipping process can be quite slow. For example, the code optimizations we will study in this lesson were able to take a sample level that previously needed 12 minutes to compile a PVS down to a process that completed in only a few seconds.

The problem with the PVS process is that it is very CPU intensive, especially on very large data sets. The key to solving this problem is a little bit of pre-processing to eliminate portals or leaves that could not possibly see one another. This allows us to bail out of certain processes much earlier because we know for sure that nothing beyond a given portal can be seen. For example, imagine that a portal had 1000 portals behind it. If we could tell for sure, before we even started doing the PVS clipping, that

nothing behind this portal could ever be seen, then we could eliminate thousands of clipping and splitting operations and shave minutes or possibly hours off the compilation time.

As briefly discussed earlier in this section when we reviewed the system components we intend to develop, we are going to write some functions that will be called prior to the actual recursive anti-penumbra clipping process that forms the core visibility processor. These routines will be used to trivially reject portals and leaves that we know for sure cannot be seen before the process even begins based on their orientations to one another. However, before we can look at this code, we need to understand the concept of *portal flow*.

17.6.5 Portal Flow

Currently, the problem with our master portal array (populated during the portal generation process) is that although we know which two leaves a portal resides in, we have what is in effect, a two-way portal. The fact that the portal is 'owned' by both leaves can lead to problems during the recursive process. To illustrate this, imagine that you have a portal that acts as the doorway between leaves 5 and 6 and that our PVS calculator uses this portal to step from leaf 5 into leaf 6 during the recursive clipping process. Once we reach leaf 6, we will then need to loop through and traverse though each of Leaf 6's portals. But we need to avoid going back through the portal we just entered so that we do not get caught in a recursive loop. Obviously this issue in and of itself is trivial and easily solved using flags or other such concepts, but remember that we also want to be able to reject portals from our process *before* we even begin our recursive PVS calculations. The easiest way to do that and to solve our recursion problem simultaneously is to use *one-way portals*.

One-way portals allow us to instantly discard any portals from consideration that point in opposite directions or that are situated behind the current source portal being processed. In short, they allow us to determine, prior to the core recursive clipping process, which portals cannot possibly be viewed from each other portal, thus minimizing the amount of testing that will need to be done in the core processor. It is very important that we remove every test we possibly can from the core recursive procedure as it is going to be executed many thousands of times for each source portal. Moving such tests out of the main clipping process can literally shave hours off the final compile time.

A one-way portal is a portal that will only allow entrance or exit in one direction. In our example, we may have a portal that leads from Leaf 5 to Leaf 6, but that portal cannot be used to step from Leaf 6 back into Leaf 5. It resides only in Leaf 5 and will not be considered one of Leaf 6's portals. One-way portals are set up so that, while they are 'owned' by one leaf only, they also contain a variable describing the Neighbour Leaf. The OwnerLeaf is the leaf in which the portal resides and the Neighbour Leaf is the Leaf that the portal leads into when stepped through from the owner leaf. The normal of the one-way portal points into the neighbour leaf, so we can enter through the back of a portal and come out in the neighbour. This is an important point to remember so it bears repeating: **a portal directs the recursive flow from the owner leaf into the neighbour leaf**.

From a practical perspective this means that we will have to have twice as many portals as we originally generated in our master portal array. Using our previous example of a portal owned by both leaves 5 and 6, we will now have to duplicate it so that there are actually two one-way portals instead of one bi-

directional portal. One portal is assigned to Leaf 5 and directs flow into Leaf 6, and that the other will be assigned to Leaf 6 and direct flow into its neighbour leaf, Leaf 5. The result is that we have two portals at the same location in 3D space which direct flow in opposite directions (into the respective neighbour). The generation of this one-way portal array will be the first step taken in the PVS calculation process.

In order to do this we are going to need a new portal structure (we will call it PVSPortal for the sake of our placeholder code discussions) that will be used to represent these temporary one-way portals for the life of the PVS calculation. We must copy over the portals from the original portal list, duplicate them, and then adjust them to fit with the new data requirements required by the PVS calculator. Since each original portal will now be represented by two PVS portals, we will need to create a new array that has enough room for NumberOfOriginalPortals * 2 elements.

Here is our new portal structure.

```
class CPVSPortal
{
public:

    // Constructors / Destructors for this Class
             CPVSPortal( );
    virtual ~CPVSPortal( );


    // Public Variables for This Class
    UCHAR             Status;              // The compilation status
    UCHAR             Side;                // Which direction does it point
    long              Plane;               // The plane on which this portal lies
    long              NeighbourLeaf;       // Leaf into which this portal flows
    long              PossibleVisCount;    // The size of the PossibleVis array
    UCHAR            *PossibleVis;         // "Possible" visibility information
    UCHAR            *ActualVis;           // "Actual" visibility information
    bool              OwnsPoints;          // Does this own the points ??
    CPortalPoints    *Points;              // The vertices making up this portal

};
```

Although the actual source code that we use will be discussed thoroughly in the accompanying workbook, understanding this structure and its members is necessary before examining the code snippets that we will discuss in this chapter. Indeed understanding this structure will play a critical role in understanding the whole compiler works. Therefore, we will discuss the members of this portal structure now. We will review them in a different order to how they are listed in the above structure, in a way that will make the description of each easier to follow.

CPortalPoints *Points

The first member is a pointer to CPortalPoints structure (shown below). This is a simple container for the vertex list and number of vertices. Actually, it is derived from a CPolygon and in a sense is the actual portal itself. We derive from CPolygon so that we can give the portal polygon some new members that pertain to the portal clipping process. The class includes both a Split and Clip function that allows us to split/clip the portal's points to the plane.

```
class CPortalPoints : public CPolygon
{
public:

             CPortalPoints( );
             CPortalPoints( const CPolygon * pPolygon,
                            bool Duplicate = false );
    virtual ~CPortalPoints( );


    CPortalPoints *     Clip( const CPlane& Plane, bool KeepOnPlane );

    virtual HRESULT     Split( const CPlane& Plane,
                               CPortalPoints * FrontSplit,
                               CPortalPoints * BackSplit);


    bool                OwnsVertices; // Are the vertices owned
    CPVSPortal         *OwnerPortal;  // Points to parent portal ;)

};
```

The OwnsVertices Boolean will be difficult to explain at the moment since we have not seen how the clipping process works. For now, let us just touch on it lightly. We know that when performing the recursive clipping process for a given source portal, we will want to clip generator portals to the anti-penumbra. However, we must remember that we will not want to delete the actual master CPVSPortal created at the start of the process as this will be needed by other source portals. So at each recursive step copies of the generator portal are created (from the original) and are passed through the clipping process. If any of these copies get clipped, new temporary portals will be created which own their own vertices and the original copy (created from the master CPVSPortal) will be deleted (just as we always delete the original polygon when it is split). However, to save memory in places where clips do not happen, copies can share the vertex array with the parent they were copied from and as such, new vertices are only generated when the portal copy actually gets clipped. However, at the end of the recursive procedure the copies are discarded. If the copy has not been clipped, then it will still share a vertex pointer with the master CPVSPortal it was copied from and as such, the copy's destructor must know under these circumstances not to delete the vertex array shared with the original PVS portal. That is what the OwnsVertices flag indicates. If it is set to false then it means that the portal is sharing its vertex array with another portal (to conserve memory) and as such, when it is deleted, its vertex array should not be freed from memory.

The OwnerPortal points to the CPVSPortal structure that owns this CPortalPoints structure.

bool OwnsPoints

When we create the CPVSPortal array will create two one-way portals from every original two-way portal. Because these portals will be geometrically exactly the same, with the exception that they have swapped Owner/Neighbor leaves, it would be wasteful if each duplicate stored its own copy of the portal polygon (CPortalPoints). Therefore, each duplicate pair will share the original vertex winding of the original polygon. As an example, this means that while N original portals will be converted into N*2 CPVSPortal structures (i.e., one-way portals), only N portal polygons (CPortalPoints) will be in

memory. As with the OwnsVertices member of the CPortalPoints class, only one of the duplicates will own the CPortalPoints and the other will just share a pointer to it. Only the owner will delete the CPortalPoints object when it is destroyed.

long Plane
This is where we store the portal plane. It will be populated when the CPVSPortal array is first created using the original two-way portal. Each pair of duplicates will share this same plane as it is the plane of the node on which the portal was created. This will be the index of the node stored in the original portal's OwnerNode member.

The node plane is stored in the portal because, as we will see in a moment, during the recursive clipping process, we often have to clip the generator portal to the source portal's plane if the generator portal is spanning the plane of the source portal. This helps narrow the anti-penumbra and also allows us to avoid inverting the anti-penumbra when that generator portal becomes the next target portal. We also have to do the same in reverse and clip the source portal to the plane of the generator portal if the source portal is spanning that plane. Although we will see why this is important later, just know for now that if this step is not taken, the anti-penumbra can essentially become inverted causing endless recursive loops.

It is fairly clear then that if we may need to clip other portals to the plane of a portal, each portal will have to store the plane on which it lies so that it can be used for the clipping operation. It should be noted however that both of the duplicate portals will store the same node plane even though they are supposed to be facing in opposite directions. This is not a problem since we will also store (in the CPVSPortal structure) whether the duplicate is considered to be facing in the opposite direction to the plane normal (one of the two always will be). With this information we know when processing such a portal, that the same plane can be re-used by the clip operation. The difference would be that any fragments of other portals that lay in front of the clip plane are removed instead of fragments that lay in the plane's backspace as is usually the case.

UCHAR Side
As each pair of portals duplicated from the original source portal will share the same plane, we will store in this variable either FRONT_OWNER or BACK_OWNER indicating whether the duplicate is facing in the same direction as the plane or in the opposite direction, respectively. This will come in very useful later in the clipping process when other portals have to be clipped to that portal. If the portal has a Side member set to FRONT_OWNER then it means this duplicate faces in the same direction as the original plane normal and as such, any portal fragments that exist in its backspace should be clipped. If the duplicate has BACK_OWNER stored instead, then it means this is the portal that has had its direction flipped from the original portal and as such is facing into the opposite halfspace with respect to the plane normal. When this is the case, our code will know that any portals that have to be clipped to its plane should have the clip operation flipped so that only fragments in the plane's front space survive. We will see this being used later in the core processor.

long NeighbourLeaf
This member will store the index of the leaf that the one-way portal leads into (i.e., the leaf which its imaginary normal is supposed to be facing into). We enter the neighbor leaf from the owner leaf by passing through the back of the portal. Notice that we do not actually have to store which leaf the portal is owned by within the CPVSPortal structure. The leaf itself will store this information and we do not

need it when doing the visibility tests. This is because we will calculate a visibility list for each portal (i.e. which leaves can be seen from a portal) and the owner of the portal is not needed until the very end of the process when we loop through all of the portals in a given leaf. Just remember that a leaf can see all of the leaves that all of its portals can see. This means that a leaf's PVS is really nothing more than the union of all of its owned portals' visibility sets.

UCHAR *PossibleVis

This is an array of bytes, where each leaf is represented by one bit. Therefore one byte holds the visibility information for eight leaves for this portal. One of the first functions we will write will be called InitialPortalVis and it will be used to quickly reject any portals that cannot possibly be seen. Our goal will be to loop through every portal and check every other portal against it. If a portal is behind the current portal being processed then there is no way the current portal can see it. In this case, we set the appropriate bit in its PossibleVis array to zero, otherwise, we set it to one. At the end of this process, we will have an array of bits in each portal describing only the portals that can possibly be seen. Portals for which no portal flow could ever happen have their bits set to zero in that portal's PossibleVis array. This process is extremely cheap to execute and will allow us to reject many portals before we even start the PVS processing. Although the PossibleVis array is not very accurate, it does allow us to check if a portal can be seen during PVS calculation. If it cannot, then we can just skip it, thus saving many clipping operations and speeding up the overall process. We can think of the PossibleVis array as being a very coarse leaf visibility set for the portal which will be refined and stored in the portal's ActualVis array (described next) when it has had its PVS fully calculated.

UCHAR *ActualVis

This member is an array of bytes which will contain the actual visibility information when the whole recursive PVS process has finished for a particular portal. Remember that PossibleVis is only used to reject portals that obviously cannot be seen, but many hundreds of leaves that did pass the PossibleVis test will be clipped out during the more precise recursive process and thus are not visible from the current portal in the end. So while PossibleVis allows us to cut down on the number of portals that will need to be tested during the recursive clipping phase, ActualVis will contain an even further reduced set than is found in the PossibleVis array. It is the ActualVis array which ends up containing the final leaf visibility set for the portal. When all portals have had their ActualVis arrays calculated, the PVS calculator is complete and its final job is to loop through each leaf and union the ActualVis arrays of each portal contained in that leaf. This will describe the PVS of each leaf, which can then be packaged together into a single array and dispatched to file. This combined block of leaf PVS data is what we refer to as the PVS for the entire scene.

long PossibleVisCount

This member will contain the number of visible leaves that were found during the initial population of the portal's PossibleVis array. For example, if the PossibleVis array determined that there are 25 portals that this portal might be able to see, this member will be set to 25 for that portal. We will see later that this allows us to select the order in which we generate the actual PVS for each portal such that we start with the portal that can see the fewest leaves first. This helps speed up the process so that more complex portals (i.e., portals that have higher visibility) can borrow information from the less complex portals which may have already had their PVS calculated. This will make more sense later.

UCHAR Status
This member tells us if a portal has already had its ActualVis array calculated. You will see how to use this flag later, but just know that it will help us refine and speed up the PVS process. Remember that our goal is to design an algorithm that has as many early-out options as possible to keep our PVS calculator running at a tolerable speed.

We now have a good understanding of the structure that will be used to store our one-way portals during the PVS calculation process so we can start to examine the coding process involved in writing the PVS calculator. We will begin with the creation of the one-way portals.

17.7 Coding a PVS Calculator

Before we start looking at each function that comprises the PVS calculator in detail, let us first run through a high level overview of the functions involved and discuss when they will be called and what task they will be expected to perform. This should keep everything in perspective as we move forward.

ProcessPVS

We will assume during our code discussions that this is the only function that needs to be called after tree and portal generation. It is the top level function responsible for calling all of the other functions in the PVS calculator (described below). When this function returns, the PVS will be generated and we will have a block of zero run length encoded data that describes the visibility of every leaf in the level.

GeneratePVSPortals

This is the first function called by CalculatePVS and it is responsible for allocating our new CPVSPortal array and copying the original portals into it. This function does everything that we just talked in the last section; it takes one portal from the original master portal array, creates two opposite portals from it, and assigns them to each leaf. At the end of this function we will have a new portal array to work with (called pPVSPortals) that will be twice the size of our original bi-directional portal array. Each portal in the array only flows one-way -- into the neighbour leaf.

InitialPortalVis

This is the second function called from ProcessPVS and it is used to fill out each portal's PossibleVis array with a very approximate PVS. If portal B is possibly visible from portal A then its bit in portal A's PossibleVis array is set to 1, otherwise it is set to 0. The function first decides which portals cannot possibly see each other (e.g., portals behind each other or portals flowing in opposite directions) and builds a temporary portal visibility list for each portal. It then calls the PortalFlood function for each portal to do a rough flood through its possibly visible portals to find out which leaves they are connected to. The leaves that are reached by the flood have their bits set in that portal's PossibleVis array to 1

indicating that they are current candidates for visible leaves to the current portal and should be more closely examined when the recursive clipping process in invoked to compile the actual PVS for the portal in question.

PortalFlood

This function is called by InitialPortalVis to perform a flood fill (for each portal) from the current portal's neighbour in all directions until it hits portals which can no longer be seen (very rough approximation) by the source portal. As it flows through each portal and into the portal's neighbouring leaves, these leaves are added to the portal's PossibleVis array. The main PVS calculation process will have to examine the leaves which have their bits set to 1 in this array.

> **Note:** The functions just discussed are not actually part of the PVS calculator as such. They are used to reduce the number of portals/leaves that will have to be considered during the actual PVS process. By doing some rough approximations we can reject some portals/leaves that we can tell are not visible without having to spend time clipping or doing other expensive operations.

CalcPortalVis

This is the function that starts the true PVS process. It is called after every portal has had its PossibleVis array calculated. Basically, it loops through each portal and calculates its actual visibility set. When this function returns, all portals will have their ActualVis arrays correctly filled with their real and final leaf visibility set. For each portal this function will call the RecursePVS function to calculate the actual visibility information for that portal. Portals do not have their visibility sets calculated in order however. Instead, the GetNextPortal method is called to return the next source portal that should have its PVS calculated. This will return a portal which has not yet had its ActualVis array calculated and that has the least number of bits set in its PossibleVis array. This allows us to calculate the PVS for less complex portals first in the hope that later, when we calculate the more complex portals, we can borrow some of the visibility information from the less complex portals that it can see. This aids in speeding up the PVS calculations for those complex portals. We will see how that all works later.

GetNextPortal

When we calculate the PVS information (ActualVis) for each portal, we do not actually do so using the order that they are stored in the portal array. Instead, we use this function to return the index of the next portal that should be calculated. This function returns the CPVSPortal from the master portal array that has not yet been processed (CPVSPortal::Status = `PS_NOTPROCESSED`) and that has the lowest number of bits sets to 1 in its PossibleVis array (PossibleVisCount = lowest). The portal returned from this function (to CalcPortalVis) is fed into the recursive clipping process as the next source portal.

RecursePVS

This function is the heart of the PVS process. Once it has been called by CalcPortalVis, it calls itself recursively until the entire PVS for a given source portal has been calculated. It walks through the leaves of the tree by stepping through the portals, and at each step clips what can be seen to the doorway/portal it just stepped though. As more and more portals are stepped through, the viewable area gets smaller and smaller until eventually no other portals can be seen (and therefore no other leaves are visible) and the recursive process unwinds and a new source portal is chosen.

ClipToAntiPenumbra

Every time a new portal is stepped through, this function is called by RecursePVS to build a new set of clipping planes. These planes will extend from the source portal to the portal just passed through in the previous recursion (the target portal). This function then clips any other portals in the new leaf (generator portals) and if any generator portals survive the clip, we know that we can still see a bit more of our scene through this portal and thus have to recur again. Otherwise, we can see no more and our journey ends there.

ExportPVS

Once we have calculated which leaves every portal can see, we need to know which leaves every other leaf can see. A leaf can see everything that all of its portals can see and thus we simply have to loop through each leaf's portals and collect all of the leaves that its portals can see. Once we have created a visibility array for each leaf, we will compress it (by calling CompressLeafSet) and add it to the master PVSData array.

CompressLeafSet

This function is called by ExportPVS once the PVS for a single leaf has been calculated (by taking the union of all the visibility bits for each of its contained portals). It is passed a byte array (1 bit per leaf) of visibility information and compresses it (using zero run length encoding) into a compressed byte array that is then added to the master compressed PVS data array.

SetPVSBit / GetPVSBit

These two functions are helper functions that allow us get or set a particular bit in a byte array. Both the PossibleVis and ActualVis arrays use one bit per leaf byte arrays and we will need to toggle their bits as a leaf is determined to be visible/invisible. For example, passing in 17 as the DestLeaf parameter will set bit 2 of byte 2 to 1 in the byte array passed by the caller via the VisArray. Just remember that the array starts at bit 0 of byte 0. The PVS calculator will use these two functions to set/retrieve bit information in both the portals PossibleVis and ActualVis arrays.

Since these functions are extremely small utility functions we will cover them here, starting with SetPVSBit. If the third parameter is set to true then it means we wish to set a bit in the passed array. As there are 8 bits per byte we can easily calculate the bit in which the visibility bit for DestLeaf resides by dividing the leaf index (DestLeaf) by 8 (>>3). By ANDing the leaf index with 7 we also locate which bit in the byte in which it resides maps to that leaf. In this case we can shift 1 by the amount to left so that we have a value with the proper bit set. We then OR this value with the corresponding VisArray array byte to toggle that bit set. If the Value parameter is false then we locate the byte in which the leaf resides in the same way but NOT it with the value on the right hand side of the equals sign. This will clear the bit (if not already cleared) in the respective byte in VisArray.

```
void SetPVSBit( UCHAR VisArray[], ULONG DestLeaf, bool Value = true )
{
    // Set / remove bit depending on the value
    if ( Value == true )
    {
        VisArray[ DestLeaf >> 3 ] |=  (1 << ( DestLeaf & 7 ));
    }
    else
    {
        VisArray[ DestLeaf >> 3 ] &= ~(1 << ( DestLeaf & 7 ));

    } // End if Value
}
```

The GetPVSBit function is also extremely simple, using the same bit shifting logic. On the left hand side of the equals sign we locate the byte in which the passed leaf's visibility bit resides and on the right hand side we once again construct that same byte value that has the bit set that we are interested in. We then return the bitwise AND of the left and right hand sides, which will only return true if the same bit is set on both sides of the equals sign.

```
bool CProcessPVS::GetPVSBit( UCHAR VisArray[], ULONG DestLeaf )
{
    return (VisArray[ DestLeaf >> 3 ] & (1 << ( DestLeaf & 7))) != 0;
}
```

We are now ready to step through the code to all of the processes involved in our PVS calculator. We will examine them in the order in which they are encountered during program flow.

17.7.1 The ProcessPVS Function

This is the function that would be called from the application that wishes to generate the PVS data for the tree (after it has been compiled).

> **Note:** While the code shown throughout this textbook should give you a very good understanding of how to code your own PVS calculator, the actual code that we use in the lab project is discussed in the workbook and is part of a larger application. The textbook code is intended to give you the general flow of how and where things happen throughout the process using placeholder code. Please note however,

that the code shown here is almost identical to the code used in the lab project, with the exception of some structures names, namespaces, and simple helper functions which we do not cover here.

This is the top level function that encapsulates the calls to all other modules involved in the process. Seeing it below gives us a good understanding of the order in which things happen. Throughout all the functions discussed, access to a pointer called pTree is assumed. This is the BSP leaf tree that contains the compiled leaf data and the portal information. That is, the master portal array that was initially generated is assumed to be stored in the BSP tree and accessible via its member functions.

```
HRESULT ProcessPVS( )
{
    // Calculate Number Of Bytes needed to store each leafs
    // vis array in BIT form (i.e 8 leafs vis per byte uncompressed)
    PVSBytesPerSet = (pTree->GetLeafCount() + 7) / 8;

    // 32 bit align the bytes per set to allow for our early out long conversion
    PVSBytesPerSet = (m_PVSBytesPerSet * 3 + 3) & 0xFFFFFFFC;

    // Retrieve all of our one way portals
    GeneratePVSPortals();

    // Calculate initial portal visibility
    InitialPortalVis();

    // Perform actual full PVS calculation
    CalcPortalVis();

     // Export the visibility set to the final BSP Tree master array
    ExportPVS( pTree );

    return BC_OK;
}
```

The first thing this function does is ask the tree for its current leaf count. This is needed because we know that later, when we calculate the ActualVis and PossibleVis arrays for each portal, these will need to be large enough to store a bit for every leaf in the tree. As discussed earlier, the number of bytes we need to allocate can be found by adding 7 to the leaf count and then dividing the total number of leaves by 8 (8 bits per byte). You will recall from earlier discussions that we add 7 to force a rounding up so that an extra byte is allocated. If we did not do this, 17 leaves divided by 8 would equal 2 because of the truncation that happens when calculating integers. But in fact, we would need 3 bytes because bit 17 would be the 17^{th} bit in the array and thus the 1^{st} bit in the 3^{rd} byte. Adding 7 before the divide ensures that we always round the byte count up and have a value that correctly describes how many bytes should be allocated to contain 1 bit of visibility for each leaf.

Next we add 3 to this amount and do a bitwise AND to clear out the last two bits. What this does is describe the size of the visibility arrays that we will later allocate to be large enough so that they can be written to with a 32-bit long pointer. As you will see shortly, we will access the visibility arrays with a pointer to type long (for speed), so we have to make sure that we pad out the end of the array with spare bytes so that it ends on a 4-byte boundary. For example, imagine that we allocated three bytes and then tried to write to the first byte with a four-byte pointer (long *). In this case, we would accidentally write

to the fourth byte, causing a page fault. Therefore, if we need 22 bytes, we will allocate 24 bytes instead; the two bytes at the end will serve as padding only and will never be used by us.

> **Note:** We will assume that the result of this calculation (PVSBytesPerSet) is located in a scope from which all functions that we will later call will have access to it during the PVS calculation process.

Once done, we call the function briefly described above to carry out the various tasks of the PVS calculator. GeneratePVSPortals is called first which will build the array of one way CPVSPortal structures that will be used by the PVS calculation process. When this function returns we will then call the InitialPortalVis function which will perform some very fast portal/portal rejection techniques and build the PossibleVis array for each portal. This describes which leaves every portal can see in this quick and cheap broad phase visibility rejection function. When this function returns, we will call the CalcPortalVis function which will perform the actual core clipping process and generate the actual PVS (ActualVis) for each portal. When this function returns every portal in the level will have had its leaf PVS generated and stored in its ActualVis array. We then call the ExportPVS method which uses the portal visibility information to build the visibility information for each leaf and compress it into a final master PVS data array. This is the array that would be saved to disk and loaded by the run-time component.

Let us now examine each of these functions one at a time, leading ultimately to a full understanding of how a PVS calculator works and how one can be implemented.

17.7.2 The GeneratePVSPortals Function

In this function we will create the array of CPVSPortal structures that are used by the PVS calculator. Remember that this array will be twice as large as the portal array currently generated and stored in the BSP tree. Each two-way portal will now be turned into two one-way portals which will be stored side by side in this new array (elements N and N+1 in the CPVSPortal array). The duplicated portals will not allocate their own vertices, they will store pointers to the vertices of the original two-way portals contained in the BSP tree.

The function first retrieves the number of two-way portals and allocates an array of pPVSPortals structures which is twice this size. In this first section of code (and throughout the rest of the functions we examine), we will assume that pPVSPortals is a container of CPVSPortal structure pointers. You can see that we use the container's Resize method to inform it to make room for the needed elements.

```
HRESULT GeneratePVSPortals( )
{
    ULONG i, p, PortalCount;

    PortalCount = pTree->GetPortalCount();

    // Allocate enough PVS portals to store one-way copies in PVS portal array.
    pPVSPortals.resize( PortalCount * 2 );

    for ( i = 0; i < PortalCount * 2; i++ )
    {
```

```
       // Allocate a new portal
       pPVSPortals[i] = new CPVSPortal;
    } // Next Portal
```

Notice above that once we have allocated the PVSPortal pointer array, we loop through every element in that array and assign its pointer a newly allocated CPVSPortal structure. At the end of the above section of code we will have allocated every CPVSPortal structure we need to duplicate all the original portals.

In this next section we will enter the main loop that will fetch every original two-way portal from the original list of portals in the BSP tree and copy it into two one-way portals in our new CPVSPortal array. Inside the loop we first fetch the portal from the tree's list via an assumed tree method called GetPortal which will return the index of a portal stored in the tree. This value is stored in the local variable pBSPPortal. We then allocate a new CPortalPoints structure which will contain the portal polygon (remember that this is derived from CPolygon) that we wish to copy.

```
    for ( i = 0, p = 0; i < PortalCount; i++, p+=2 )
    {
       // Retrieve BSP Portal for easy access
       CPortal * pBSPPortal = pTree->GetPortal(i);

       CPortalPoints *pp = AllocPortalPoints( pBSPPortal, false );
```

The AllocPortalPoints function requires some explanation even though it is a simple housekeeping method and its code will not be shown here in the textbook. Basically it creates a new CPortalPoints structure and either copies its vertices from the passed portal (1st parameter) or if the second parameter is false, it simply stores a pointer to the passed portal's vertex data in the CPortalPoints member. That is, because we have passed false as the second parameter, the returned CPortalPoints structure will not have had its own geometry (vertex data) allocated, but will instead just assign its vertex data pointer to point at the vertex data of the original two-way BSP portal. The false parameter also instructs the allocation function to set the CPortalPoints::OwnsVertices member to false so that the CPortalPoints destructor will know that when it is destroyed, it should not delete its vertex data because it does not own it. This vertex data is owned by the original two-way polygon in the tree.

Now we will fill out the first duplicate in the CPVSPortal array. Variable p starts at zero at the start of the loop and is incremented by 2 for each iteration. This is because we add two duplicate portals to CPVSPortals with each iteration of this loop. Therefore, variable p always contains the index into the CPVSPortals array where these two duplicates should be placed. The first duplicate should be stored in pPVSPortals[p] and the second (opposite facing portal) in pPVSPortals[p+1].

```
       // Create link information for front facing portal
       pPVSPortals[p]->Points = pp;
       pPVSPortals[p]->Side   = FRONT_OWNER;
       pPVSPortals[p]->Status = PS_NOTPROCESSED;
       pPVSPortals[p]->Plane  = pTree->GetNode( pBSPPortal->OwnerNode )->Plane;
       pPVSPortals[p]->NeighbourLeaf = pBSPPortal->LeafOwner[ FRONT_OWNER ];
       pPVSPortals[p]->OwnsPoints    = true;

       // Store owner portal information (used later)
       pp->OwnerPortal = pPVSPortals[p];
```

In the above code we first store the newly allocated CPortalPoints structure pointer in the first duplicated portal's Point member and set the Side member to FRONT_OWNER. Remember that this portal is going to represent the duplicate that is facing in the same direction as the node plane on which the portal was created. We also set its status to PS_NOTPROCESSED which tells us that this portal has not yet had its PVS calculated. Then we fetch the node plane on which the original portal was created (recall that this is stored in the portal's OwnerNode member). Since this duplicate is going to be the one that has the leaf in front of the node plane as its neighbor, we fetch this leaf's index from the portal as well. This index will be stored in the original portal's LeafOwner array in the FRONT_OWNER element. Finally, we set this new portal's OwnsPoints member to true because it is the PVS portal (out of the two duplicates we are creating in this iteration) that will take responsibility for the CPortalPoints structure we allocated for it. The other duplicate will still store a pointer to it, but it will have a value of false as its OwnsPoints member. Again, this allows both duplicates to share a single CPortalPoints structure without them both trying to delete this same object when the portals are released from memory. Only the owner of the CPortalPoints object (the one-way portal we have just created in the above code) will delete this array. At the bottom of the above section of code you can see that we store a pointer to the one-way portal in the CPortalPoints::OwnerPortal member. Specifying the owner in this way will come in handy later.

> **Note:** Remember that the CPortalPoints structure is really just a CPolygon structure containing a few extra member variables (such as the OwnerPortal member). It is this member of the CPVSPortal that contains the actual geometry for the portal.

In the next section of code we create the second duplicate one-way portal using the original portal currently being copied. We then store it next to the previously created one-way portal in the CPVSPortal array. Notice that this portal stores a pointer to the same CPortalPoints structure (after all, these portals describe the same physical portal polygon), but this time we record that this portal is owned by the leaf in front of the plane and as such its neighbor is the portal situated behind the node plane. That is, this portal's plane is assumed to have a normal facing into the back leaf of the original node. However, notice that we store the same node plane in both cases. This is fine, because we know that this portal points into the backspace of the node plane and we will invert the plane normal of this duplicate prior to clipping any portals to its plane (shown later).

```
        // Create link information for back facing portal
        pPVSPortals[p+1]->Points        = pp;
        pPVSPortals[p+1]->Side          = BACK_OWNER;
        pPVSPortals[p+1]->Status        = PS_NOTPROCESSED;
        pPVSPortals[p+1]->Plane = Tree->GetNode(pBSPPortal->OwnerNode )->Plane;
        pPVSPortals[p+1]->NeighbourLeaf = pBSPPortal->LeafOwner[ BACK_OWNER ];
        pPVSPortals[p+1]->OwnsPoints    = false;
   } // Next Portal

     // Success!!
     return BC_OK;
}
```

Notice also that this portal does not own the CPortalPoints structure even though it stores a pointer to it. As mentioned, this is so that it will not try to delete this object when the portal is deleted. We will leave that task to the destructor of the first duplicate we created above it in the code.

That is the first stage of the PVS calculator covered. When this function returns we will have an array of one-way portals. Every original portal from the BSP tree will have been duplicated into two one-way portals that face in different directions and have different neighbor leaves. Pairs of duplicates will share a single CPortalPoints structure that contains the portal geometry and each CPortalPoints structure will contain a pointer to the vertex array in the original two-way portal from which it was cloned.

We saw earlier that after the GeneratePVSPortals function returns program flow back to the ProcessPVS function, we then issue a call to the InitialPortalVis function. This is the function that is responsible for doing the cheap visibility test for each portal. It builds each portal's PossibleVis array. This function will naturally be covered next.

17.7.3 The InitialPortalVis Function

The next function called by ProcessPVS is InitialPortalVis. Its job is to build a rough approximation of leaves that might be visible from each portal. This will allow us to reject as many leaves as possible before entering the recursive PVS process. If we did not include this step, our PVS would take much longer to calculate. Because we now have portals that only flow one way, it makes it quite easy for us to tell that some portals cannot possibly be visible from another portal.

For example, if a portal flows to the right (i.e. its neighbour leaf is to the right of the node plane) then we know immediately that any other portals that lay to the left of this portal cannot be seen. In short, we know that Portal A cannot possibly see Portal B when B is behind it. Consequently it is also unable to see into any of the leaves that Portal B leads to. If we know beforehand that a portal cannot possibly see a leaf, then we do not have to go through the lengthy process of clipping all of the portals in that leaf during the PVS process. This is especially true when we know that all of those portals would be completely clipped away anyway. This will save us a great deal of time.

Figures 17.53 through 17.55 depict portals that may be trivially rejected by our function. For each portal, InitialPortalVis will loop through every other portal in the list and perform two tests. If a portal passes both of the tests, then it can (potentially) be seen by the current portal. Otherwise, we know for certain that it cannot.

The InitialPortalVis function will loop through each portal and first construct a temporary byte array describing which portals can be seen by the current portal (the 'portal visibility list'). For example, if portal 2 has the 5^{th} byte in this temporary array set to 1, it means portal 2 can possibly see the 5^{th} portal. This test will be performed for every other portal in an inner loop which, upon completion, will have set the corresponding bytes in this array to 1 for any portals in the PVS portal array which are considered initially visible. This provides us with the first major step forward -- which portals can be seen by the current portal being processed. Any bytes set to zero in this array indicate portals that the current portal cannot possibly see through the back of (i.e., the direction of visibility through a portal) because it is either facing towards the current portal or is situated behind it.

Once we have the temporary array describing which portals the current portal can see, we will call the PortalFlood function (more on this shortly) and pass it this portal visibility array so it can then calculate which *leaves* might be seen by the current portal. The leaves which may *possibly* be visible will have

their bits set in that portal's PossibleVisibilitySet (CPVSPortal::PossibleVis array) which will be used later when we calculate the PVS and create each portal's ActualVisibilitySet (CPVSPortal::ActualVis). So this function will first start off by looping through each portal. In this loop an inner loop will be instantiated that will loop through every other portal in the scene and we will run two tests on it to see if the test portal is potentially visible from the current portal being processed.

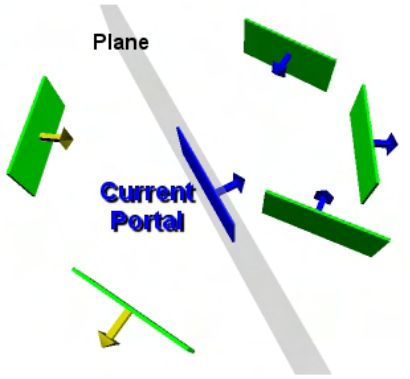


Figure 17.53

The diagram we will use to demonstrate these tests is shown in Figure 17.53. The central blue portal is the current portal that is about to have its initial (possible) visibility calculated. The plane on which this portal lies is also shown. In this image there are five other green portals (with normals indicating their flow direction into their neighbor leaves) and it is each of these we will need to test against the current portal. Basically we need to determine if flow can ever exist between the current portal and each of these portals based on the portal orientations and spatial relationships to one another.

The first test we will perform is whether or not the test portal is situated behind the current portal. By 'behind' we mean that it is situated behind the portal with respect to the portal's flow direction. If it is, then no flow could possibly happen from the current portal to the test portal because a portal only has visibility through the back of the other portal (and thus, into that test portal's neighbor leaf). If the test portal is behind the current portal, this cannot possibly be the case. Such portals are rejected immediately and do not have their corresponding bytes set to 1 in the portal visibility list being temporarily compiled for the current portal. In Figure 17.54 we can see that with this simple test, two of the five possible portals are rejected since they lay behind the portal's plane. Thus, they are located in the opposite halfspace with respect to the current portal's view direction. It should be noted however that these portals are only rejected if they lay *completely* behind the current portal. If the test portal is spanning the current portal's plane then it will still be marked as visible at this time because at least some of that test portal will be visible to the current portal (the section in the portal's frontspace).

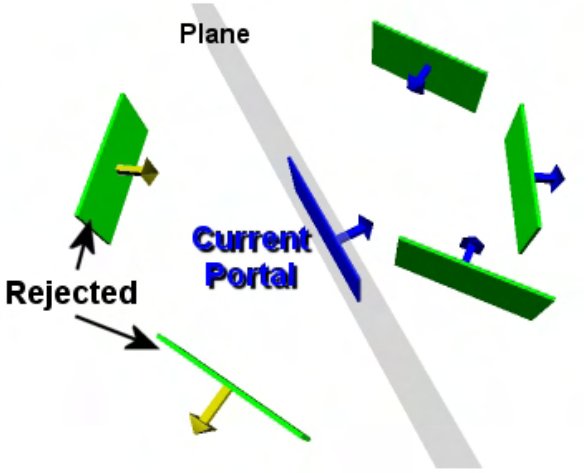


Figure 17.54

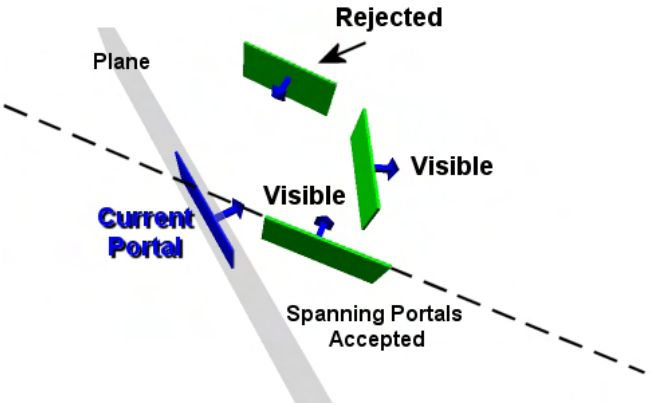


Figure 17.55

If the test portal is not situated completely behind the current portal's plane then it will have survived the first test and at this time is still considered as possibly visible. We then feed it into the second test that classifies the current portal's location with respect to the test portal. With our one way portals, a given portal can only see through another portal if the direction in which the current portal is facing flows through the back of the test portal. Therefore, as we can see in figure 17.55, this cannot possibly be the case if the current portal is completely in front of the test portal's plane. If it is, then it must mean that the current portal's normal and the test portal's normal are facing into each other and therefore, we have visibility flow in opposing directions. When this is the case the test portal is also skipped and is not added to the portal visibility list for the current portal. In Figure 17.55 we can see that this test has rejected yet another portal. This leaves us with just two portals in the current portal's visibility list that are flagged as potentially visible. Note once again than in this second test, the test portal is only rejected if the current portal lies completely in the test portal's frontspace. If the current portal is spanning the test portal's plane then it means at least some small portion of the current portal must be able to see the back of the test portal. Therefore, some small amount of visibility still exists between the current portal and the back of the test portal.

Once every portal has been pumped through these two tests for the current portal being processed, we will have collected (in a temporary byte array) the visibility status of each portal with respect to the current portal being processed. This is then used to perform a flood fill from the current portal through any visible portals in this array. This flood fill will start off at the neighbor leaf of the current portal and will recursively walk through any portals contained in that leaf which also have their visible bytes set to 1 in the portal's temporary portal visibility array.

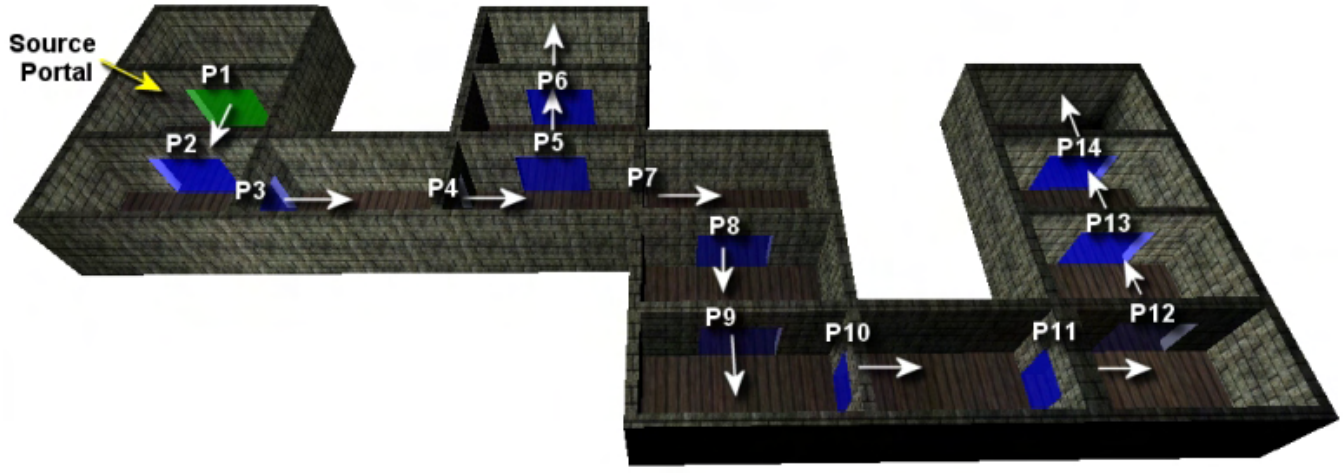


Figure 17.56

In Figure 17.56 we see some example geometry and the portals contained in each leaf. Portal P1 is assumed to be the portal currently having its visibility set calculated. The portal flood procedure will

start off in the neighbor leaf of the current portal and the leaf will have its bit set in the CPVSPortal::PossibleVis array. We will then iterate though any portals in that leaf, stepping through them into their neighbor leaves, which will have their bits set in the portal's PossibleVis array as well. Now, if this was all we were to do, the recursive procedure would simply flood through every portal in the level making every leaf 'wet', adding it to the current portal's (P1) possible visibility set. However, prior to performing this procedure to calculate leaf visibility, we first performed our two tests against each portal that allowed us to build an array of portal visibility information from the current portal P1. As discussed, any portals that are located behind the current portal or that have normals facing in the opposite direction to the current portal are considered non-visible. During the flood fill, when we enter a new leaf and test each of its portals (which always flow out of the leaf into the neighbor), we will only step through that portal and into its neighbor leaf if that portal's byte has been set to 1 in the source portals portal visibility array.

For example, imagine we are calculating the possible leaf visibility set for portal P1 in Figure 17.56. The white arrows in the diagram show the flow as we traverse through the portals from leaf to leaf. We can see for example that we would first add the neighbor leaf to P1's possible visibility set and would then flow through its portal P2, into its neighbor leaf. This leaf would also be added to the portal's visibility array. We would continue to step though P3 and P4, adding every leaf we visit (leaves that get 'wet' by the flood) to the possible visibility array in the source portal. When we pass through portal P4, something interesting happens; its neighbor leaf has two portals (P5 and P7) so we should recur through them both. However, the normal for portal P5 is facing in the opposite direction to the normal of the source portal P1 and therefore, this portal would have its corresponding byte in P1's portal visibility array set to 0, informing us that no visibility flow can happen between these two portals. This is actually fairly intuitive when you look at their spatial arrangement. Because portal P5 is not flagged as visible, we skip this portal and do not step into it. Thus we do not visit P5's neighbor leaf or the neighbor leaf for portal P6. We have just rejected two leaves from the source portals PossibleVis array.

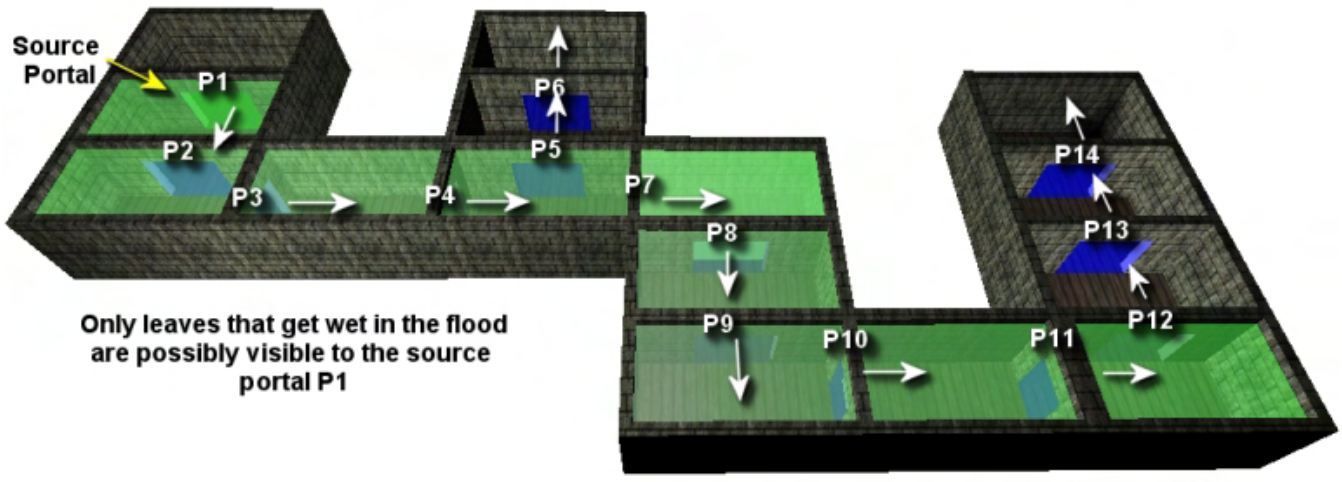


Figure 17.57

Back in the neighbor leaf of P4 we can see that after skipping portal P5 we recur through the other portal in that leaf P7 and we continue to flow through a combination of portals right up to and including portal P11, setting the bit for each leaf we walk through to 1 in the source portal's possible visibility array.

Once in the neighbor leaf of P11, we find it has one portal (P12) but that its visibility byte in the source portal's portal visibility array is set to zero because it flows in an opposite direction to the source portal (P1). Therefore we never step though this portal or into any of the leaves and portals it leads to (P13 and P14). As you can see, even in this simple example, we have now rejected 5 leaves from the source portal's PossibleVis array. When we get around to calculating the actual PVS (ActualVis array) for the source portal, we know that these portals and their neighbor leaves should never be traversed into since they cannot possibly be visible from the source portal.

At the end of the InitialPortalVis function that we are about to write, every portal in the scene will have had its PossibleVis array calculated and the bits of any potentially visible leaves set to 1. Although this is a very crude approximation of the portal's PVS, it does serve as a foundation upon which we can build the actual PVS for each portal (ActualVis).

> **Note:** It is quite common during development to turn off the generation of the actual PVS calculation and simply use the PossibleVis array for each portal to calculate the PVS. While this represents only a very rough approximation of what it and is not visible, and should never be used in commercial game code, it does allow the developer to quickly compile and run the game many times with some degree of PVS (even if it is too generous) without having to wait for the full PVS calculator to generate its much tighter visibility set.

Let us now look at the code that does everything we have discussed in this section. At the end of this function (InitialPortalVis), the PossibleVis array for every portal will have been calculated and will contain a very approximate leaf visibility set.

The code first allocates a byte array large enough to store a visibility status byte for each one-way portal in the level. This code assumes the existence of a GetPVSPortalCount function which will return the number of PVS portals (one-way portals) in the portal array. This temporary array will be re-used in each iteration of the outer loop to store the visibility status of every other *portal* in the scene. We then set up the outer loop to loop through each PVS portal that we generated earlier and fetch a pointer to that portal and its plane.

```
HRESULT InitialPortalVis()
{
    CPortalPoints  *pp;
    ULONG           p1, p2, i;
    CPlane          Plane1, Plane2;
    UCHAR          *PortalVis = NULL;
    CPVSPortal     *pPortal1, *pPortal2;

    // Allocate temporary portal visibility buffer
    PortalVis = new UCHAR[ GetPVSPortalCount() ];

    // Loop through the portal array allocating and checking
    // portal visibility against every other portal
    for ( p1 = 0; p1 < GetPVSPortalCount(); p1++)
    {
            // Retrieve first portal for easy access
            pPortal1 = GetPVSPortal( p1 );

            // Retrieve portal's plane
            GetPortalPlane( pPortal1, Plane1 );
```

pPortal1 is the current portal we are processing the preliminary visibility information for, and the function GetPVSPortal is assumed to be a function that will return a pointer to the CPVSPortal pointer in the requested position in the CPVSPortal array. What is vitally important to note is the use of the GetPortalPlane method. We might think that this function is unnecessary, considering the portal structure stores the plane, however you will recall that both duplicate portals created from an original portal store the same plane, even though they are supposed to be pointing in different direction. The GetPortalPlane method tests to see if the passed portal has its Side member set to BACK_OWNER. If so, then this is the duplicate portal that is pointing in the opposite direction to the normal of the node plane on which the portal was originally created. In this instance, the GetPortalPlane function will negate the plane normal and distance before returning the node plane. If the passed plane is the FRONT_OWNER, then the node plane is returned unchanged. Essentially, this function makes sure that we get the node plane back facing in the correct direction corresponding to the one-way portal.

Since pPortal1 (the current portal being processed) will have its CPVSPortal::PossibleVis member pointing to a byte array that contains the preliminary leaf visibility set for this portal (1 bit per leaf), we had better allocate the memory for this array next and initialize it to zero. You will recall that PVSBytesPerSet was calculated earlier (in ProcessPVS) to contain the number of bytes that will be needed to store 1 bit per leaf for each leaf in the tree.

```
            // Allocate memory for portal visibility info
            pPortal1->PossibleVis = new UCHAR[PVSBytesPerSet];
            ZeroMemory( pPortal1->PossibleVis, PVSBytesPerSet );

            // Clear temporary buffer
            ZeroMemory( PortalVis, GetPVSPortalCount() );
```

Notice in the above code how we also clear the PortalVis byte array that will be used to store a byte per portal in the initial portal visibility phase. Remember, the contents of this array will be calculated first, so we know exactly which other portals could possibly be visible from the source portal (pPortal1). This same array will be reused during each iteration of this loop to store the portal visibility bytes for each portal that we process.

With the arrays initialized, it is now time to loop through every other portal and perform the two tests discussed earlier. Only if the test portal passes both tests will it have its corresponding byte set to 1 in the PortalVis array. The first part of the loop tests to see if the current test portal is the same as the current source portal, and if so, just skips the tests. We also fetch the test portal and its plane using the same GetPortalPlane function described earlier to make sure that the plane normal is facing in the same direction as the portal.

```
            // For this portal, loop through all other portals
            for ( p2 = 0; p2 < GetPVSPortalCount(); p2++)
            {
                // Don't test against self
                if (p2 == p1) continue;

                // Retrieve second portal for easy access
                pPortal2 = GetPVSPortal( p2 );
```

```
                // Retrieve portal's plane
                GetPortalPlane( pPortal2, Plane2 );
```

At this point we have the source portal (pPortal1) and the test portal (pPortal2) and our first test will be to loop through each vertex in the test portal and classify it against the source portal. As soon as we find a vertex in the test portal that is not in front of the current portal, we break from the vertex loop. This will tell us outside the loop that the test portal is not completely behind the source portal and cannot be rejected from the possible visibility set just yet.

```
                // Test to see if any of p2's points are in front of p1's plane
                pp = pPortal2->Points;
                for ( i = 0; i < pp->VertexCount; i++)
                {
                  if ( Plane1.ClassifyPoint( pp->Vertices[i] ) == CP_FRONT ) break;
                } // Next Portal Vertex

                // If loop reached end, there were no points in front so continue
                if ( i == pp->VertexCount ) continue;
```

However, if at the end of the loop the loop variable ‘i’ is equal to the vertex count of the test portal, we know that none of its vertices were ever found to be in front of the source portal and as such we can skip the rest of this loop. No visibility can exist through this portal from the source portal as it is situated in the source portal’s backspace. The test portal will not have its visibility byte set to 1 in the PortalVis buffer.

However, if some of the vertices in the test portal were found to be in front of the source portal, we must perform the second test in the hope that it may yet be rejected. In this next test we test each vertex in the source portal against the plane of the test portal and break as soon as we find a vertex that lives in the test portal’s back space. If the vertex loop does not exit prematurely, then it means that the source portal is total in the test portal’s frontspace and thus they must be facing each other. Since portal flow can only happen from the source portal through the back of the test portal, the source portal must not be able to see through the test portal. In this case, the test portal is skipped and does not have its corresponding byte in the PortalVis array set to 1.

```
                // Test to see if any of p1's portal points are Behind p2's plane.
                pp = pPortal1->Points;
                for ( i = 0; i < pp->VertexCount; i++)
                {
                  if ( Plane2.ClassifyPoint( pp->Vertices[i] ) == CP_BACK ) break;
                 } // Next Portal Vertex

                // If loop reached end, there were no points in front so continue
                if ( i == pp->VertexCount ) continue;

                // Fill out the temporary portal visibility array
                PortalVis[p2] = 1;

            } // Next Portal 2
```

Outside the inner loop we will have filled out the PortalVis buffer with information describing which portals are visible from the source portal and which ones are not (and therefore will stop the flood). The

last thing we do for the current source portal is call the PortalFlood function which will perform a flood fill starting from the source portal's neighbor leaf through all visible portals. Any leaves that get 'wet' by the flood will have their visibility bit set to 1 in the source portal's PossibleVis array. Notice that before we call this function, we set the source portal's PossibleVisCount to zero as we have not yet found any visible leaves. This will be incremented in the portal flood function every time a new leaf is traversed into. We will look at the PortalFlood function next. Notice how the PortalVis buffer we have just compiled for the current source portal is passed into the function as it contains the information that is used to block the flood at non-visible portals.

```
            pPortal1->PossibleVisCount = 0;
            PortalFlood( pPortal1, PortalVis, pPortal1->NeighbourLeaf );

        } // Next Portal

    // Clean up
    if (PortalVis) delete []PortalVis;

    return BC_OK;
}
```

At the bottom of the function, after exiting the outer loop, the PossibleVis array for every portal in the level will have been calculated and we will be ready to start calculating the real PVS for each portal with the anti-penumbra clipping we discussed earlier. Before the function returns, the temporary memory that was allocated for the portal visibility information is discarded. We no longer the need the information since each portal now stores which leaves are considered visible by the portal flood. It is this set that will be refined in the next phase.

17.7.4 The Portal Flood Function

Because we have calculated which portals can see each other in the calling function (InitialPortalVis) and we have passed that visibility buffer into this function, it becomes trivial to perform a flood fill out from the current portal to see which leaves it can get to before it hits portals that it cannot see (i.e.,. stopping the flood). We are certainly all familiar with flood fills used in paint applications, where the color fills an area within some specified set of bounds. We can think of this function as basically doing the same thing.

We start by passing in the portal we are calculating and the portal's neighbor leaf. Once in that leaf, we can check whether any portals in that leaf are visible from the source portal, and if so, walk through that portal into its neighbor leaf. We keep doing this recursively until we hit a place where the portals are no longer visible from the source.

This is a very small recursive function and very simple compared to some of the more complex recursive function we have written in the past. The function is initially called from the InitialPortalVis function and is passed the source portal whose PossibleVis array we wish to populate, the portal visibility buffer containing the visibility of each portal with respect to the source portal, and the

neighbor leaf of the source portal, which is the leaf in which the flood will begin. We will show the first half of the function first and then discuss it.

```
void CProcessPVS::PortalFlood( CPVSPortal * SourcePortal,
                               unsigned char PortalVis[],
                               unsigned long Leaf )
{
    CBSPLeaf * pLeaf = pTree->GetLeaf( Leaf );

    // Test the source portals 'Possible Visibility' list
    // to see if this leaf has already been set.
    if ( GetPVSBit( SourcePortal->PossibleVis, Leaf ) ) return;

    // Set the possible visibility bit for this leaf
    SetPVSBit( SourcePortal->PossibleVis, Leaf );

    // Increase portals 'Complexity' level
    SourcePortal->PossibleVisCount++;
```

The first thing this function does is fetch the leaf pointer from the BSP tree using the passed leaf index. It then uses the GetPVSBit utility function to determine whether the leaf we have just stepped into already has its bit set to 1 in the source portal's PossibleVis array. If it is, then it means we have visited this leaf before during the flood and should take no action and just return. Continuing to step through the portals in this leaf under these circumstances could allow us to get stuck in a recursive flow loop. For example, we move from Leaf A into Leaf B into Leaf A and so on forever (or at least until we get a stack overflow). This provides loop protection by saying, "we have been here before, so do not process this leaf and its portals again". Finally, in the above code you can also see that, provided we have not been here before, we use the SetPVSBit utility function (discussed earlier) to set the bit that corresponds to the passed leaf index in the source portal's PossibleVis array. Notice that since we have stepped into a visible leaf, we increment the portal's PossibleVisCount member so that at the end of the flood (for the current source portal) we will know how many leaves were considered visible.

With this leaf added to the source portal's possible visibility set, our final task is to loop through any other portals owned by this leaf and step through them into their neighbor leaves. The leaf itself stores the indices of the two way portals that were originally created, so we will have to bear that in mind when using the leaf portal indices to fetch the one-way portals from the CPVSPortal array. This is no problem however, since we know that all we have to do is multiply the original indices by 2 to get the position in the one-way portal array where the two duplicates were made. We can then use that index to fetch the one-way PVS portal and, if this is the duplicate that has the current leaf as its neighbor leaf, we know it is not the correct duplicate. We are after the second duplicate in this instance, where its owner is equal to this leaf, so we increment the portal index we calculated and grab it from the PVSPortal array.

```
    // Loop through all portals in this leaf (remember the portal numbering
    // in the leaves match up with the originals, not our PVS portals )
    for ( ULONG i = 0; i < pLeaf->PortalCount; i++)
    {
        // Find correct portal index (the one IN this leaf (not Neighbouring))
        ULONG PortalIndex = pLeaf->PortalIndices[ i ] * 2;
        if ( GetPVSPortal( PortalIndex )->NeighbourLeaf == Leaf ) PortalIndex++;
```

At this point we have the current portal in this leaf that we wish to step through into its neighbor leaf, but if this portal does not have its byte set to 1 in the passed PortalVis buffer, we know that this portal was found not to be visible from the source portal and we can skip it. This halts the flood in that direction. If this is not the case, we continue to flood through the portal into the neighbor leaf. Below is the remainder of the function.

```
        // If this portal was not flagged as allowed to pass through, skip it
        if ( !PortalVis[ PortalIndex ] ) continue;

        // Flood fill out through this portal
        PortalFlood( SourcePortal,
                     PortalVis,
                     GetPVSPortal( PortalIndex )->NeighbourLeaf );
    } // Next Leaf Portal
}
```

This function returns program flow back to the InitialPortalVis function when all leaves that could be flooded into through all visible portals have been reached and flagged as visible in the portal's PossibleVis array. When the InitialPortalVis function returns back to the ProcessPVS function, the PossibleVis array will have been populated for each portal. The master function next invokes the actual PVS processor -- the CalcPortalVis function.

Below we see the main parent function ProcessPVS and highlight (in bold) the functions we have examined so far. As you can see, the CalcPortalVis function is the next process we need to execute and when this function returns, the actual PVS for every portal will have been created.

```
HRESULT ProcessPVS( )
{
    // Calculate Number Of Bytes needed to store each leafs
    // vis array in BIT form (i.e 8 leafs vis per byte uncompressed)
    PVSBytesPerSet = (pTree->GetLeafCount() + 7) / 8;

    // 32 bit align the bytes per set to allow for our early out long conversion
    PVSBytesPerSet = (m_PVSBytesPerSet * 3 + 3) & 0xFFFFFFFC;

    // Retrieve all of our one way portals
    GeneratePVSPortals();

    // Calculate initial portal visibility
    InitialPortalVis();

    // Perform actual full PVS calculation
    CalcPortalVis();

     // Export the visibility set to the final BSP Tree master array
    ExportPVS( pTree );

    return BC_OK;
}
```

17.7.4 The CalcPortalVis Function

CalcPortalVis calculates each portal's ActualVis array. We can think of the ActualVis array as a localized per-portal PVS. The ActualVis array will be a subset of the PossibleVis array calculated in our last lesson, because many of the 1 bits in PossibleVis will be set to 0 in ActualVis when we determine that those leaves are not truly visible.

This is the top-level function that begins the recursive process of stepping through the various leaves in the tree and marking them as visible. The first thing we need to do is to loop through each portal in our array and calculate the ActualVis for it. Figure 17.58 provides some insight into the recursive clipping process that will be involved in this function and its recursive helper function.

We loop through each one-way portal in our array and start a recursive process that uses this portal as the current source portal. The source portal's neighbor leaf becomes the first target leaf. The first target leaf can always be seen from the source portal because the source portal is connected to it directly. We then loop through each portal in the target leaf and build an anti-penumbra between the source portal and the target portal. (Remember that each portal in the target leaf is called a target portal.)

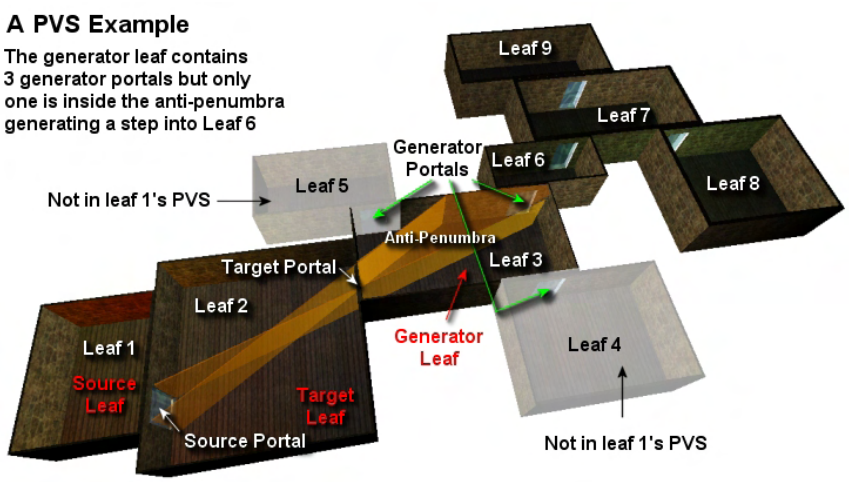


Figure 17.58

If a target portal is visible from the source portal, then we know that we must be able to see into the target portal's neighbor leaf (which we call the generator leaf). We then check all portals in the generator leaf and clip them to the anti-penumbra. If any of these generator portals survive the clipping process, it means that the source portal can see the generator portal. This in turn tells us that the source portal must also be able to see into the generator portal's neighbor leaf.

> **Note:** A generator portal is named as such because if it is visible from the source portal, it 'generates' another recursion.

This is where the recursive part comes into play. Once we have found a generator portal that is visible from the source portal, we repeat the process. Only this time, the generator portal becomes the new target portal and we rebuild the anti-penumbra between the source portal and the new target portal (the old generator portal). The neighbor leaf of the old generator portal (which is now the new target portal) becomes the new generator leaf and the whole routine happens again. We check all the generator portals in the new generator leaf, clip them to the new anti-penumbra, and if any portals survive, then they

become target portals themselves in the next recursion and the anti-penumbra is rebuilt. We then recur into the next leaf, and so on.

When we conclude that a portal is visible and step through it into its neighbor leaf, we set that leaf's bit in the current source portal's ActualVisBits array to 1. This tells us that this leaf is indeed visible from the source portal. This same recursive process is done for every portal in our array (i.e., every portal in our array will have its turn at being the source portal and will have its ActualVisBits array calculated in this manner).

This function uses a data structure that we have not yet discussed, but which will make more sense once we look at the next function (RecursePVS). Because we will need to recur through this function many times, it will be helpful to have a data structure where we can pass information from the previous recursion to the next. For example, when the RecursePVS function calls itself, the current generator portal needs to be passed into the next recursion as the target portal. We also need to be able to access the source portal at all times, regardless of what instance of the function we are in, so that we can set the bits for each visible leaf we encounter. So before we look at the code to the CalcPortalVis function, we will quickly discuss the PVSDATA structure:

```
typedef struct _PVSDATA             // Structure to hold pvs processing data
{
   CPortalPoints   *SourcePoints;   // Current source portals points
   CPortalPoints   *TargetPoints;   // Current target portals points
   CPlane           TargetPlane;    // The target's plane
   UCHAR           *VisBits;        // Visible Bits being calculated
} PVSDATA;
```

This structure is mainly used so that we can access data about the generator portal that was selected in a previous iteration of the recursive function. Since the previous generator portal becomes the target portal in the next recursion, this allows us to find a generator, fill out the information about it in one of these structures, and then pass on this structure to the next recursion so that we can access and use it as the target portal. There are also some other fields in here which are not explicitly connected to the target portal.

The SourcePoints and TargetPoints pointers will point to the actual vertices for the source and target portals. Earlier, we discussed how a generator portal is clipped to the anti-penumbra, and if any of it survives, we recur again, sending the new clipped portal into the next iteration as the new target portal. We do not actually have to send the generator portal itself into the next recursive call; we only need to send the new clipped points (the vertices). Therefore, at each recursive step, TargetPoints will hold the new clipped target portal points that survived the previous function call (when it was the generator portal). The clipped TargetPoints will then be used to rebuild the anti-penumbra planes between the source portal and the new target portal. We will discuss this later, but we also clip the source portal during the recursive process. This is done to help refine our PVS and get as close as possible to a perfect visibility set. So not only will we clip the generator to the anti-penumbra, but we will also build a second anti-penumbra from the generator portal to the target portal and clip the source portal to these planes. This means that at each step, the anti-penumbra will get smaller at both ends, and will increase our chances of shaving a few more polygons out of the portal's PVS. For now, there is no need to worry about all of this since we will discuss it all in detail later.

TargetPlane is the plane of the target portal. Remember that the target's plane is just the previous generator portal's plane. Finally, the BYTE pointer VisBits will be used in a very interesting way to speed up our PVS calculator. We will discuss this when we look at the RecursePVS function.

The first section of the function is shown below.

```
HRESULT CalcPortalVis()
{
    ULONG   i;
    PVSDATA PVSData;

    // Clear out our PVSData struct
    ZeroMemory( &PVSData, sizeof(PVSDATA) );

    // Lets process those portal bad boys!! ;)
    for ( i = 0; i != -1; i = GetNextPortal() )
    {
        CPVSPortal * pPortal = GetPVSPortal( i );
```

The first thing we do in the above code is instantiate a PVSDATA structure and initialize it to zero. Then we do something a little bit different; instead of just looping through each portal in the array and calculating the PVS for them one at a time, we enter a while loop and call a function called GetNextPortal. This function returns the portal index of the portal to be calculated next, until all portals have been processed. When this happens, it will return -1, which means that we will have calculated the PVS for each portal in the list.

The obvious question is why not just loop through the portal array in linear order and go through them one at a time? Although we will not fully know the answer to this question until we look at the next function, it turns out that we can speed up our PVS calculations if we calculate the PVS for the least complex portals first. The least complex portals in the list are the portals which have the fewest number of bits set in their PossibleVis arrays. In other words, we choose portals that can see the least amount of leaves first. If we calculate the quickest ones first, then there is a good chance that when we come to calculate more complex portals (i.e., portals that can see more leaves and will take longer to calculate), we may be able to reuse the information from the less complex portals to speed things up.

The GetNextPortal function iterates through the CPVSPortal array and returns the index of the portal with the fewest number of bits set in its PossibleVis array which has not already been used. When a portal has had its PVS calculated, we set the portal's Status member to PS_PROCESSED so that it is never returned from the GetNextPortal function again.

Next we fill in our PVSData structure so that PVSData.SourcePoints holds the vertex list for the source portal. The source portal is the portal we have just chosen to calculate the PVS for (selected by GetNextPortal). We will need access to the source portal's points because we may need to clip them, as we will discuss later. This structure will also store a pointer to the source portal's PossibleVis array which will be needed during the recursive clipping process, along with the plane of the source portal. This data structure will be passed into the RecursePVS function where much of the main work happens.

```
            // Fill our our initial data structure
            PVSData.SourcePoints    = pPortal->Points;
```

```
            PVSData.VisBits         = pPortal->PossibleVis;
            GetPortalPlane( pPortal, PVSData.TargetPlane );
```

As discussed, this data structure's primary job is to provide access to the previous generator portal and the source portal as we step from one recursion to another. The generator portal in one recursion will become the target portal at the next recursion of the function and this data is passed between instances of the function using this structure. At first it may seem strange to send in the plane of the source portal as the TargetPlane member in the data structure. But remember that on our first call to RecursePVS, we will not yet have a target portal. The RecursePVS function will need to detect if it is in its first recursion and behave differently when this is the case. We will use this plane on the first recursion just to check that the new target portal is not on the same plane as the source portal. Essentially, by storing the source plane in the very first call, we tell the recursive method that this is the first time through, so do not try to build an anti-penumbra; just choose a target portal and step through it. This is done automatically since we ignore portals that are co-planar to the source portal. Since it is wasteful to have a whole new plane variable called SourcePlane just to hold a variable for the top level recursion, we decided to use (or misuse if you prefer) TargetPlane to hold the source portal's plane for this one time. This allows us to find valid target portals by testing against it.

Next we allocate memory for the current source portal's ActualVis array. After RecursePVS returns, this array will hold the real PVS for this portal. We can then set this portal's status flag to PS_PROCESSED so that it is ignored by future calls to GetNextPortal.

```
            // Allocate the portals actual visibility array
            pPortal->ActualVis = new UCHAR[ m_PVSBytesPerSet ];

            // Set initial visibility to off for all leaves
            ZeroMemory( pPortal->ActualVis, m_PVSBytesPerSet );
```

Next we call the RecursePVS function, which is the main recursive processing routine that creates the anti-penumbras and calculates the actual PVS (ActualVis array) for the current source portal. The RecursePVS call takes three parameters, shown below.

```
            // Step in and begin processing this portal
            RecursePVS( pPortal->NeighbourLeaf, pPortal, PVSData );

            // We've finished processing this portal
            pPortal->Status = PS_PROCESSED;

        } // Next Portal

    // Success!!
    return BC_OK;
}
```

The first parameter is the index of a leaf in the leaf array. Normally, this leaf index will be the neighbor leaf of a generator portal that we have just accepted and stepped through. This means that on the next recursion, the generator portal will become the target portal and this leaf will become the new generator leaf. We can then loop through the portals in this new leaf to try to find additional visible generator portals.

The next parameter we pass in is a pointer to the source portal (i.e. the portal we are about to calculate the PVS for). This pointer will remain constant through each recursion. This allows us to access the source portal's ActualVis array so that we can set the bits for visible leaves to 1 as we find them.

The final parameter is the address of the PVSData structure. Some of the members are unused at this moment because it is the source portal that we are passing in. For each recursion after the first one, this data structure will hold the TargetPoints for the new target portal. The target portal will be the generator portal from the previous recursion (or what was left of it after clipping it to the anti-penumbra) and the TargetPlane will contain the plane of this portal as well.

Finally, notice that after the RecursePVS function returns, the PVS for the current source portal will have been created, so we set the portal's Status flag to PS_PROCESSED. This will let the GetNextPortal method know that this portal has been processed already and should not be returned as a source portal again. By the time the outer loop of the above function exits, the PVS for every portal in the level will have been computed and all the hard work done.

Before we examine the RecursePVS function which is arguably one of the most complex we have discussed so far (although not that large) we will take a look at the GetNextPortal method which was used by this function to select the portals for PVS calculation, in order of complexity.

```
ULONG GetNextPortal( )
{
    CPVSPortal * pPortal;
    long PortalIndex = -1, Min = 999999, i;

    // Loop through all portals
    for ( i = 0; i < GetPVSPortalCount(); i++ )
    {
        pPortal = GetPVSPortal(i);

        // If this portal's complexity is the lowest and it has
        // not already been processed then we could use it.
        if ( pPortal->PossibleVisCount < Min && pPortal->Status == PS_NOTPROCESSED)
        {
                    Min = pPortal->PossibleVisCount;
                    PortalIndex = i;

         } // End if Least Complex

     } // Next Portal

     // Set our status flag to currently being worked on =)
     if ( PortalIndex > -1) GetPVSPortal( PortalIndex )->Status = PS_PROCESSING;

     // Return the next portal
     return PortalIndex;
}
```

17.7.5 The RecursePVS Function

This function is the heart of the PVS calculator. It is called from CalcPortalVis to generate the PVS for a given source portal. RecursePVS and its helper function, ClipToAntiPenumbra, represent the recursive engine that is responsible for crawling through all of the leaves (starting at the source portal's neighbor leaf) and marking them as visible in the source portal's ActualVis array. Since this is a fairly complex function, we will take it one step at a time.

```
HRESULT RecursePVS(     ULONG Leaf,
                        CPVSPortal * SourcePortal,
                        PVSDATA & PrevData )
{
    ULONG           i,j;
    bool            More;
    ULONG          *Test, *Possible, *Vis;

    PVSDATA         Data;
    CPVSPortal     *GeneratorPortal;
    CPlane          ReverseGenPlane, SourcePlane;
    CPortalPoints  *SourcePoints, *GeneratorPoints, *NewPoints;

    // Store the leaf for easy access
    CBSPLeaf * pLeaf = pTree->GetLeaf( Leaf );

    // Mark this leaf as visible
    SetPVSBit( SourcePortal->ActualVis, Leaf );
```

The first thing we do (after setting up some local variables) is create a pointer to the target leaf whose index is passed in as the Leaf parameter. We do this by passing the index into the BSP tree's GetLeaf method which is assumed to return the relevant leaf pointer in this placeholder code.

The leaf index passed in as this parameter describes the target leaf which ***was*** the neighbor leaf of the previous generator portal that was found in a previous instance/recursion of the function. The first time this function is called by CalcPortalVis, we will not yet have a previous generator portal, so this leaf index will actually describe the source portal's neighbor leaf. This is the leaf at which the flood begins for the source portal. Regardless of whether this is the first instance or not however, this leaf will be the target leaf in this instance of the function and is the leaf we have arrived at after stepping though a previous generator portal (or the source portal if this is the first recursion). Remember that every generator portal that passes the visibility test will be walked through by this function so that we arrive at the generator portal's neighbor leaf (which is the leaf we get to if we walk through the generator portal). This function will then call itself again passing in the generator portal's neighbor leaf as the Leaf parameter so that on the next recursion, it will become the target leaf. The target leaf is always the leaf used by this function to search for generator portals contained within it in the hopes of stepping through into their neighbor leaves.

If we have managed to get to this leaf, then it must be visible from the source portal, so we set this leaf's bit to 1 in the source portal's ActualVis array as you can see above. This means that when this function is called, the Leaf passed in will have already passed the visibility test in the previous recursion. For example, when we first call this function from the CalcPortalVis function, we pass in the source portal's

neighbor leaf. We know that it is going to be visible from the source portal because it is the leaf that the source portal leads into. Because of this, we need to set its bit to 1 in the source portal's ActualVis array. If this function has been called from a previous recursion, then this leaf is the leaf that we have stepped into after walking through a generator portal.

Our next task is to configure the local PVSDATA structure that we instantiated on the stack at the top of the function. This is like the PVSDATA structure passed into this function from the caller and we will fill it with details about any generator portals that we discover in this recursion. For example, if we find a generator portal that is visible in the current target leaf (the Leaf parameter), we will transfer its details into this data structure and pass it to the next recursion of the function (much as we did when we first called the function from CalcPortalVis). This means that every time the function is called (except for the first time), the PrevData parameter to this function will contain a data structure containing the generator portal found in the last recursion and any information we need pertaining to it. This previous generator portal can then be used as the target portal in this instance of the function.

```
    // Allocate our current visibility buffer
    Data.VisBits = new UCHAR[ m_PVSBytesPerSet ];

    GetPortalPlane( SourcePortal, SourcePlane );
```

The first thing we do when setting up the PVSDATA structure that will be passed into the next recursion of the function (assuming visible generators are found in the current leaf we are processing) is allocate memory for storing some visual information. Recall that in the CalcPortalVis function we set the first data structure's VisBits pointer to point at the source portal's PossibleVis array. Every time we find a new generator portal, we will be able to refine this PossibleVis array as we work our way from portal to portal. We use this array to give us the opportunity for an early-out option, as it will constantly keep track of the leaves we have visited and leaves we do not have to visit.

> **Note:** Please remember that the GetPortalPlane method is assumed to return a plane that matches the direction of the passed portal.

For example, imagine we that have a target portal which can see five portals to the left and five portals to the right. The target portal might possibly be able to see all of these portals (and all of their neighbor leaves) and therefore may have all of these bits set to 1 in its PossibleVis array. Now assume that in the generator leaf (which the target portal leads into) there are two generator portals, one on the left leading to the other portals on the left, and one on the right leading to the other portals on the right. The left generator portal might be able to see many more portals to the right than the target portal can, but also, the generator portal would not be able to see any of the portals to the left of the target portal because these would be behind the generator portal's plane. Therefore, we are only interested in leaves that can be seen from the target portal when looking through the current generator portal. In other words, we do not want to know about leaves that the generator portal can see but the target portal cannot see. We also do not want to know about portals that the target portal can see and the generator portal cannot see. We can only see leaves from the source which both the target and the generator portals can see because this is the current combination of portals that the source portal is supposed to be looking through.

If we do a logical AND with the generator portal's PossibleVis array and the target portal's PossibleVis array, we manage to zero out all of the bits from the source portal's PossibleVis that cannot been seen using both the target and generator portal combination we are currently processing. This means that

with every recursion of the function along a given path, we manage to zero the bits out of leaves that cannot possibly be seen using the current combination of target/generator portals.

This also means that at every level of recursion, we can check the bits in the data structure's (Data) VisBits array against all of the bits in the source portal's ActualVis array, and if there are any set in Data.VisBits which are not set in the ActualVis array of the source portal, then we still have some possible visible leaves to test. Otherwise, if there are no bits set in Data.VisBits which are not already set in the source portal's ActualVis array, it means that there is nothing visible in this leaf which we have not seen before and we can just skip this generator portal. This will allow us to avoid visiting the same portals and leaves unnecessarily when using different combinations of target and generator portals.

No doubt some of this can be a bit confusing, as there is a lot to consider. We will look at some diagrams in a moment to help clear up some of these concepts and clarify why we have set up this PVSData structure and why we have allocated an empty VisBits array for it.

Our next step is to create some local pointers of type unsigned long. The first pointer (called Possible) points to the temporary VisBits array that we have just allocated and have yet to populate, and the second points to the source portal's ActualVis array.

```
    // Store data we will be using inside the loop
    Possible    = (ULONG*)Data.VisBits;
    Vis         = (ULONG*)SourcePortal->ActualVis;
```

The reason we do this is that we will be checking every bit in the VisBits array against every bit in the source portal's ActualVis array to determine whether or not we can see anything new in this leaf. If there are many leaves in the level, then this will be an expensive loop. Normally, because these are byte arrays we would have to check every byte from one array with every byte in the others. By setting up these two long pointers, we can compare four bytes at time, which will make the testing four times faster (this will make a more sense in a moment when we see these pointers being used).

It may appear that we are currently missing something. After all, we have not yet used the temporary Data.VisBits array that we allocated above. All we have done is allocated enough memory to hold a bit for every leaf in the BSP Tree. This is the piece that will fall into place once we find a next generator portal to process.

With our initial data structures set up, it is now time to loop through each portal in the target leaf (the leaf index passed into this function) and check whether we can see any of them. In each iteration of this loop we test a single generator portal and, if found to be of interest, we will perform some anti-penumbra clipping on it in the hopes of stepping through it, if it should survive the clipping process.

```
    // Check all portals for flow into other leaves
    for ( i = 0; i < pLeaf->PortalCount; i++ )
    {
        // Find correct portal index (the one IN this leaf (not Neighbouring))
        ULONG PortalIndex = pLeaf->PortalIndices[ i ] * 2;
        if ( GetPVSPortal( PortalIndex )->NeighbourLeaf == Leaf ) PortalIndex++;

        // Store the portal for easy access
        GeneratorPortal = GetPVSPortal( PortalIndex );
```

Just inside the start of the generator portal loop, we fetch a pointer to the generator portal stored in that leaf that matches the index of the current loop variable. We are essentially just looping through the portal indices stored in the current leaf, but we must remember that these portal indices reference the original two-way portal array. Therefore, we must multiply the index by two to get the index into the one-way CPVSPortal array where the two duplicates of the original portal we are processing in the leaf are stored. PortalIndex in the above code is therefore the index of the first of the two duplicates in the CPVSPortal array that were created from portal[i] in the current target leaf. Since the first one-way portal of the pair in the array might not be the one owned by this leaf (it might have this leaf as its neighbor), we must test to make sure that its neighbor leaf is not equal to the target leaf index. If it is, then this one-way portal is not owned by the target leaf and therefore, the one following it in the array must be the one we want. In this case, we just increment the portal index and feed it into the GetPVSPortal function which is assumed to return the CPVSPortal pointer stored at the relevant index in the one-way portal array.

Next we check whether the target portal (the previous generator portal stored in the PrevData structure passed from the previous recursion) can see the leaf that this generator portal leads to. If the target portal does not have the generator portal's neighbor leaf set in its PossibleVisBits array, then it cannot see this portal and we continue on to the next one.

```
        // We can't possibly recurse through this portal if it's neighbour
        // leaf is set to invisible in the target portals PVS
        if ( !GetPVSBit( PrevData.VisBits, GeneratorPortal->NeighbourLeaf ) )
           continue;
```

While it might seem logical to think that a target portal can always see a generator portal, this is not necessarily the case. A target portal will not be able to see its own duplicate portal (the portal in the same space that faces the opposite way) so this prevents our stepping back into the leaf we just entered through and getting caught in an endless recursive loop between two leaves. Remember that the target portal's duplicate portal will be owned by the current leaf, so we must ignore it. Because a portal will never be able to see its duplicate (because it is ON_PLANE with itself and therefore would not have had its bit set to 1 in its PossibleVis array by the InitialportalVis function), this test allows us to avoid this problem and reject any other generator portals in the current leaf that are ON_PLANE with the target portal. Notice above however, that we are using PrevData->VisBits for this comparison instead of using the target portal's PossibleVis array. This temporary VisBits array is important and we will need to discuss it in more detail in a moment. Remember however that during the first call to the function, PrevData->VisBits will be a pointer to the source portal's PossibleVis array but will change as we recur.

The next section of code is a little complicated but it provides a significant performance optimization to the recursive engine by offering early outs. Study the following code for a few moments and get a feel for the variables used and what seems to be happening.

```
        // If the portal can't see anything we haven't already seen, skip it
        if ( GeneratorPortal->Status == PS_PROCESSED )
            Test = (ULONG*)GeneratorPortal->ActualVis;
        else
            Test = (ULONG*)GeneratorPortal->PossibleVis;

        More = false;
```

```
        // Check to see if we have processed as much as we need to
        // this is an early out system. We check in 32 bit chunks to
        // help speed the process up a little.
        for ( j = 0; j < m_PVSBytesPerSet / sizeof(ULONG); j++ )
        {
            Possible[j] = ((ULONG*)PrevData.VisBits)[j] & Test[j];
            if ( Possible[j] & ~Vis[j] ) More = true;

        } // Next 32 bit Chunk

        // Can we see anything new ??
        if ( !More ) continue;
```

In order to understand this code we need to take some time to discuss the temporary Vis array in more detail. While the previous code snippet demonstrates exactly how it is used, even this small snippet of code can be extremely difficult to visualize.

Earlier we discussed the fact that if we AND the generator portal's PossibleVis array with the target portal's PossibleVis array, we end up with only the bits set in the local Vis array that *both* portals can possibly see. This makes sense because we are only interested in knowing how much of the generator portal's visibility set we can see while looking through the target portal. This has the potential to significantly reduce the generators portal's bit set, and in turn, the leaves that we have to process. What is even better is that it is entirely possible that the generator portal itself may have already had its ActualVis array (its own PVS) array already calculated if this portal was selected before the current source portal. In this case we can use the ActualVis array for the AND operation with the source portal's PossibleVis array and shave even more bits out of this temporary array, since we know that the generator portal's ActualVis array will be a lot more accurate than its PossibleVis array. As the prior code snippet illustrates, we can first check the generator portal's status flag, and if it has already had its ActualVis array calculated (status = PS_PROCESSED) then we can setup a pointer to this array (test) and use it instead of the (not as good) alternative of using the generator portal's PossibleVis array.

Being able to use the ActualVis array will be quicker because it means that many of the leaves will have already been tested and set to zero because they could not actually be seen from the generator portal when it had its ActualVis calculated. This time-saver is the reason why we used the GetNextPortal function to return the least complex portals first in the previous function we discussed. If we calculate the least complex portals first, then hopefully a more complex one can use much of this information during traversal to reject many leaves and zero them out immediately.

Now that we have a pointer (test) to either of the generator portal's visibility arrays (actual or possible), we can bitwise AND this array with the target portal's PossibleVis array (stored in PrevData.VisBits and passed from the previous recursion) and store the result in the Data.VisBits array that we allocated in this function a little earlier. Remember that the Data.VisBits array is pointed to by the long pointer called Possible. This means that at the end of the loop, this array will store only the bits that can be seen by both the target portal *and* the generator portal together. Notice that because we use long pointers, we can do the loop in 32-bit chunks for speed and therefore, compare 32 leaves per iteration.

> **Note:** The Data.VisBits (pointed to by Possible) buffer compiled here is passed into the next recursion as the PrevData.VisBits parameter and as such this process is ongoing and the PrevData.VisBits member passed to each recursive step acquires more zeros with each portal that we pass through.

Now that we have this array, our next job is to test it against the source portal's ActualVis array (pointer to long pointer Vis) which contains all the visible leaves we have found so far. If there are any bits set in Possible which are not set in Vis, then it means that we have to carry on because there are some leaves through this generator portal which have not yet been examined. If there are no bits set in Possible that are not set in Vis, then we can skip this portal because all of the bits have already been set in the source portal's ActualVis array. There is no point in stepping down that path and performing a myriad of anti-penumbra clipping operations only to flag leaves as visible that we have already visited.

Note: In short, the above code is saying, "First give me only the leaf bits that can be seen though both the target and generator portals since this is the combination of portals through which the source portal is currently looking". The resulting bits are stored in Possible (i.e., Data.VisBits). We then essentially say "Now test if this buffer of leaves that can be seen by both the target portal and the generator portal contains any leaves which we have not already flagged as visible in the source portal's ActualVis array". If not, there is nothing new to see through this combination of portals so we do not step through the generator portal.

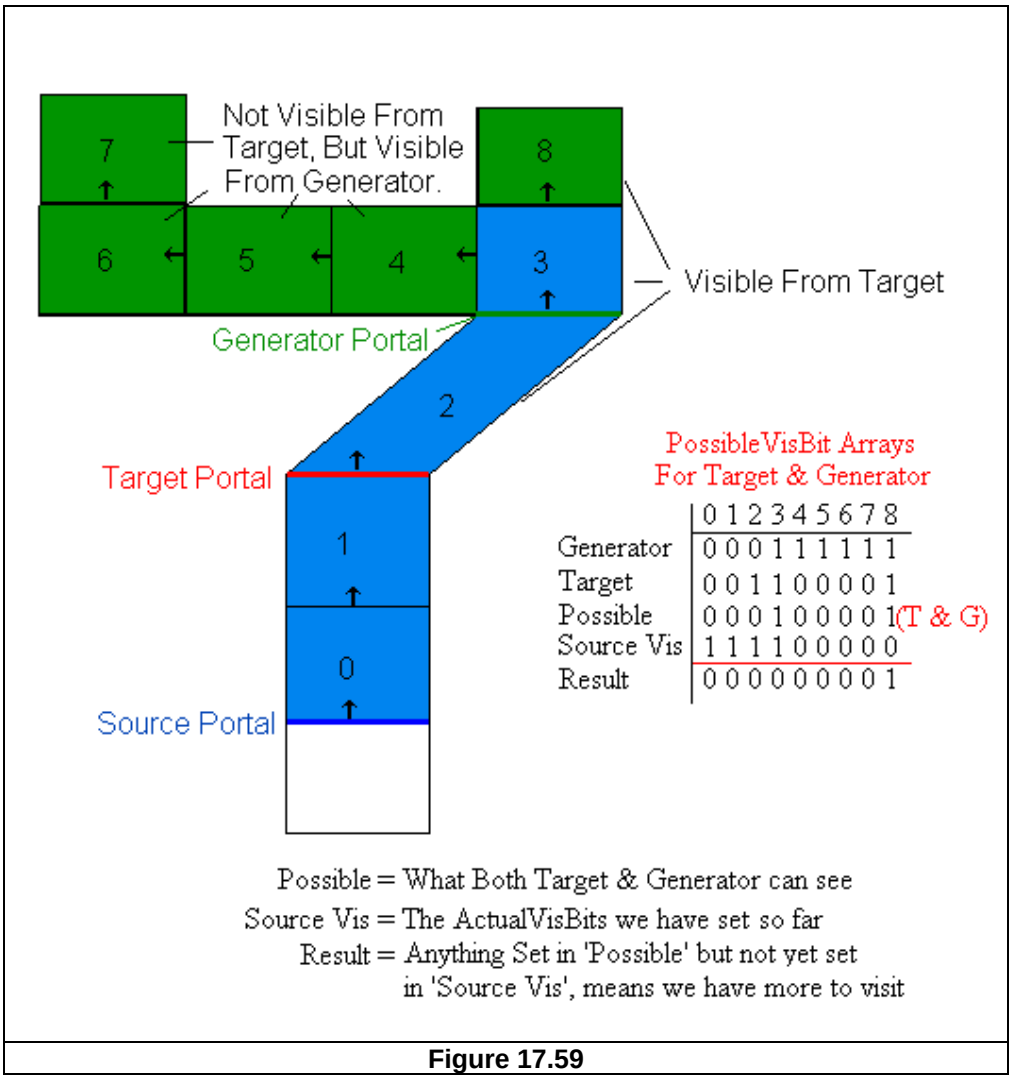


Figure 17.59

Figure 17.59 demonstrates a situation where we are midway through calculating the ActualVis array for the source portal. You can see that we have already stepped through leaves 0, 1, 2 and 3 and we have flagged them as visible (see 'Source Vis' in the bit table). Our next job is to see if there is anything visible in leaf 3 (something that we have not yet seen) that we should see.

First of all, the generator portal can see leaves 3, 4, 5, 6, 7, and 8, so it has these bits set in its PossibleVis array. This is shown in the bit table and labeled 'Generator'. The target portal can see leaves 2, 3 and 8, and as such, will have these bits set in its PossibleVis array. (The target portal cannot see through portal 4 because they are both front facing.) We can see the target portal's bits in the bit table labeled 'Target'. What we do next is a bitwise AND between the generator bits and the target bits and store the result in Possible (remember that this is really Data.VisBits). This will leave us with only the bit set for the leaves that can be seen by both the target and generator portals. Because leaves 4, 5, 6 and 7 cannot be seen by the target portal, their bits are set to zero and we know we will never have to visit them through this combination. We can only see what is visible through each target portal, and those leaves are not. This allows us to zero out the leaves in the generator portal's PossibleVis array that we cannot possibly see and thus avoid visiting them.

Now that we have this array, we check it against the source portal's ActualVis array (called 'Source Vis' in the table in Figure 17.59) and if we find any bits not set to 1 that are set to 1 in the Possible array, then there may still be new things to see, so we must continue through the generator portal (providing we later find it is within the anti-penumbra). Otherwise, we can skip this portal because nothing new can be seen through the current target/generator combination.

Notice in the example that there was 1 bit (leaf 8) that was not yet set in the source portal's ActualVis array, but was set in the Possible array. This means that that we still have to visit leaf 8. By looking at the diagram, we can see that this does in fact hold true. Leaf 8 does need to be examined because it can be seen from the source portal.

That is basically what that last section of code does -- it allows us to exit early if nothing new can be seen, and avoid recursion over the same data over and over again. This is a real time-saver since the Possible array (Data.VisBits) is passed on to the next recursion of the function if a generator is found. This means that the zeroing out effect is on-going because in our example, the Possible array will be passed to the next recursion, where it will be used as the target argument. Every time the function is called, the PrevData.VisBits array passed to each function will contain more and more zeros, until we can start skipping portals and entire sections of the level, either because they have already been visited, or because they are not visible through the portals that we have traveled through up to that point. Please take a few extra moments to study the previous code and Figure 17.59 until you are sure that you understand how this works.

One other item that may be confusing is that at first glance it seems impossible that we would visit the same leaves twice (which would seem to diminish the enhancement offered by the previously discussed optimization). Figure 17.60 however demonstrates a small but quite common case where the code we just discussed will help speed things up.

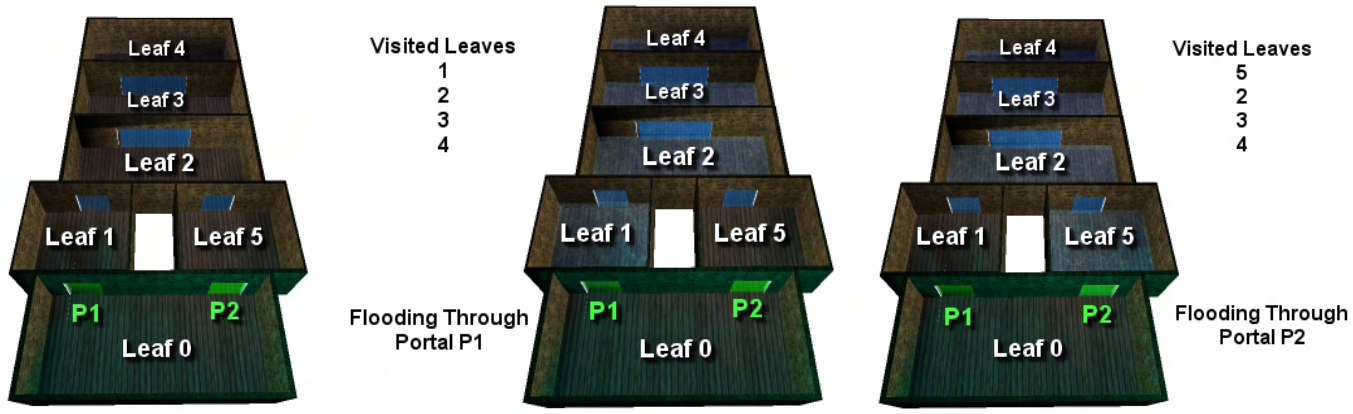


Figure 17.60

In Figure 17.60 we see that the source portal's neighbor leaf has two target portals. If we start with the portal P1 for example, we end up in leaf 1, then 2, 3, and finally 4. While traveling through each leaf, we are performing the usual clipping and testing of generator portals against the anti-penumbra. When we hit leaf 4, we have reached the end of the road because there is no leaf to see, so we abort the recursive process and unwind all the way back to leaf 0. If we repeat the process for the target portal in leaf 0 (portal P2), all is well until we exit leaf 5 and go into leaf 2. In leaf 2, we have to perform all of the same steps as before for leaves 2, 3, and eventually leaf 4, and as we can see, we are just revisiting places that have already undergone calculation and have already been determined as visible. It is not hard to imagine this example taking on a much more realistic scale (like in an actual 3D game level) where situations like the source portal in Figure 17.50 could frequently pop up and cause hundreds of leaves to be recalculated two or more times. We might find a situation where more than two target portals (like those shown in the diagram) cause thousands of redundant and unnecessary anti-penumbra clipping calculations.

> **Note:** Without these early-out cases, the PVS calculator will still work, albeit significantly more slowly. As mentioned earlier in the lesson, these modifications were able to reduce compilation time for one of our test levels from over 10 minutes to only a few seconds. This is obviously an incredible savings given the relatively small amount of code that is necessary to achieve it.

If we have made it this far in the function code without skipping the generator portal, then it is becoming more likely that the generator portal may be visible from the source portal, and thus requires full processing. Therefore, we copy the generator's plane into the Data.TargetPlane member so that if we do have to recur through the generator, this portal plane can be used as the target plane in the next recursive call. Remember that any generator portals that get accepted in this instance of the function will be stored in this data structure and passed on to the next recursion, so that we can use it as the new target portal in that instance.

```
        // The current generator plane will become the next recursions target plane
        GetPortalPlane( GeneratorPortal, Data.TargetPlane );
```

Our next test is to make sure that the generator portal is not ON_PLANE with the target portal. If it is, then they cannot see each other and we can skip this generator.

```
        // We can't recurse out of a coplanar face, so check it
        ReverseGenPlane.Normal   = -Data.TargetPlane.Normal;
```

```
        ReverseGenPlane.Distance = -Data.TargetPlane.Distance;
        if (FuzzyVectorCompare( PrevData.TargetPlane.Normal,
                                ReverseGenPlane.Normal,
                                0.001f ) ) continue;
```

Note that for this test we are using a generator plane with a reversed normal and are comparing this normal against the normal of the target portal (PrevData.TargetPlane). We do this because we need to check that the portal we have just entered this leaf through is not on the same plane as any we are about to exit this leaf through. Thus, we need to check if any of the portals in this leaf are co-planar with the portal we have just entered through. Because the portal we have just stepped through will belongs to the previous leaf that we were in, it means that its normal will be facing into the current leaf (since the current leaf is its neighbor leaf). The other portals we are checking in the current leaf are owned by the current leaf and have their normals pointing out of this leaf into their neighbor leaves. This means that even if one of the portals belonging to the current leaf is on the same plane as the portal we entered this leaf through, the normals of the two portals will be pointing in opposite directions. So we need to flip one of the portal normals for the comparison test. In order to compare the vectors, we just subtract them from each other, and if the absolute value returned is zero (with tolerance) then we have a match. This generator reversed plane will also come in handy later on as we shall see. Note that in the above code this normal compare (with tolerance) is assumed to be done by a utility method called FuzzyVectorCompare that accepts the two vectors (the normals) that we would like to compare and an epsilon value to use for the comparison (0.001 in this example). If the function returns true, the two vectors are considered equivalent and we will skip this generator portal and move on to the next one in the leaf. Thus, we will not step though it since the target portal could not possibly see through the back of it.

Our next test determines whether any part of the generator portal lies behind or is co-planar with the source portal's plane. Recall that in our InitialPortalVis function, portals that were behind other portals were not included in that portal's PossibleVis array because a portal cannot see a portal that is behind it. A generator portal spanning the plane of the source portal would still be in the source portal's PossibleVis array however and steps must be taken when this is the case to make sure that we build an anti-penumbra of the correct shape. Figure 17.61 shows an example of situation.

A source portal might be able to see a generator portal, but it might not be able to see **all** of it. In Figure 17.61 we can see that a section of the generator portal that is behind or co-planar with the source portal's plane cannot be seen by the front of the source portal and would lead to an incorrect and

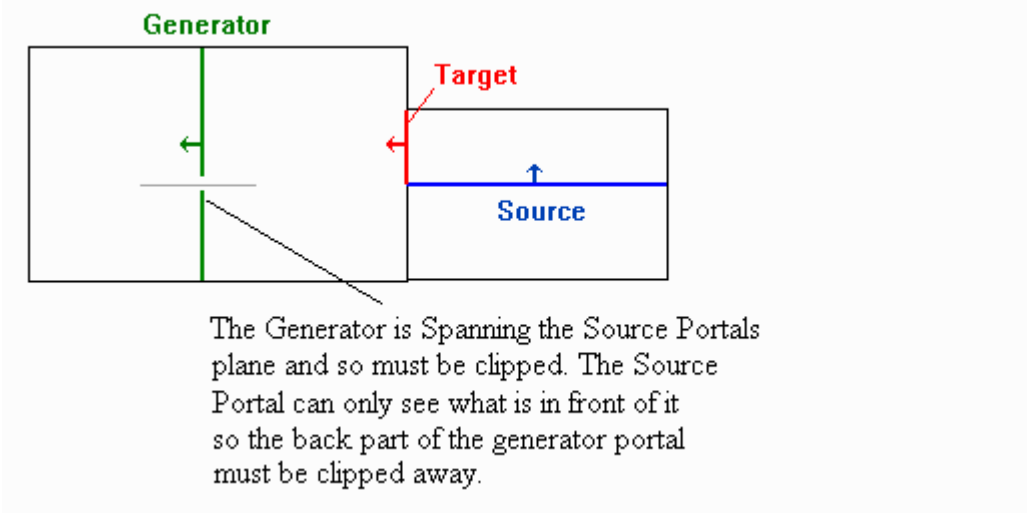


Figure 17.61

overgenerous visibility set for the current source portal. Therefore, we have to clip the generator portal to the source portal's plane so that we are only left with the front part of the generator portal.

```
        // Clip the generator portal to the source. If none remains, continue.
        GeneratorPoints = GeneratorPortal->Points->Clip( SourcePlane, false );
        if ( GeneratorPoints != GeneratorPortal->Points )
            FreePortalPoints( GeneratorPortal->Points );
        if (!GeneratorPoints) continue;
```

In the above code, we call the CPortalPoints::Clip function to clip away any part of the generator portal's geometry that is behind the source portal's plane. We are certainly familiar with how to clip and split polygons at this point in the course. We pass in the generator portal's Points and get back a pointer to a new CPortalPoints structure containing the clipped portal points. Only points in front of the plane passed into the function (SourcePlane) survive the clip and are returned. We store the result in the local pointer GeneratorPoints. The FALSE parameter simply states that we do not want portals on the same plane as the source portal to be accepted in the clip operation. It is quite important to get rid of on-plane portals as well, because when we need to build our anti-penumbra a bit later, we will completely separate the source and target portals into two separate spaces. The generator portal will become the next recursion's target portal and will be used to build the clip plane. The more important point to remember for now is that there is no way for two co-planar faces to see each other.

> **Note:** The clip function will return a new clipped copy of the portal's geometry which we can store and continue to clip recursively as we progress through the level. The original CPortalPoint structure's vertices are not deleted since they will be needed for all the other portals we wish to calculate the PVS for. When we clip any portals in this function, we are working with temporary copies that we pass through the recursive process, which are discarded at the end.

The next step we take in the above code is check to make sure that the generator points were not completely clipped out of existence. If they were, then it means that we have clipped the generator portal completely away and none of it can be seen from the source portal. In this case, the Clip function will return NULL. We will check for this condition and skip this generator portal when it happens, because it cannot possibly see the source portal in that case.

The next section of code is specifically written to handle the case when this is the first call to the function. As discussed earlier, the first time this function is called, we only pass the source portal into the data structure and thus there are no previous generator portals to use as a target portal.

When we called this function initially from CalcPortalVis, we did not give the passed data structure's TargetPoints (PrevData) member any value (it was left as NULL). Because this pointer gets assigned to the current generator portal's points later on in this function, if this member is NULL, then it must mean that this is the first time the function has been called for this source portal. In this case, the current generator portal must be in the source portal's neighbor leaf (i.e., the first target leaf), and consequently must be a target portal that is visible from the source portal. When this is so, we simply copy over the new generator portal information into this function's data structure and recur through the target portal into the target portal's neighbor leaf. This will be the first generator leaf. No clipping happens in this first case; we simply collect a target portal and step through it. It is not until we do this that we have a target leaf and a generator leaf.

```
        // The second leaf can only be blocked if coplanar
        if ( !PrevData.TargetPoints )
        {
            Data.SourcePoints = PrevData.SourcePoints;
            Data.TargetPoints = GeneratorPoints;
            RecursePVS( GeneratorPortal->NeighbourLeaf, SourcePortal, Data );
            FreePortalPoints( GeneratorPoints );
            continue;

        } // End if Previous Points
```

The above function will only be called the first time this function is called in the recursive process for each source portal. In this initial call we are not actually looking for generator portals, but are instead searching for our first batch of target portals. When the function recurs, the generator portal we have just found will be the target portal. Notice also how we pass in the portal's neighbor leaf as the next leaf to visit. This means that in the next instance of the function we will have stepped through the target portal and will be in the generator leaf. While it may seem strange or perhaps even unnecessary, our next step will be to clip the generator portal to the target portal's plane. After all, the target portal's neighbor leaf *is* the generator leaf, so technically these portals are in the same leaf. Indeed, since leaves in a BSP tree are always convex, there is no way the generator portal could ever span the target portal. Although this is true, we are using this code as a safety measure, just in case a portal has been corrupted due to floating point accumulation errors or something of that nature. Our portals should never suffer from this problem in practice, but we will play it safe anyway.

```
        // Clip the generator portal to the previous target.
        // If none remains, continue.
        NewPoints = GeneratorPoints->Clip( PrevData.TargetPlane, false );
        if ( NewPoints != GeneratorPoints ) FreePortalPoints( GeneratorPoints );
        GeneratorPoints = NewPoints;
        if (!GeneratorPoints) continue;
```

Although we have not yet discussed this, we will see later that, in addition to building an anti-penumbra between the source and target portals, we will also build an anti-penumbra between the generator and target portals and clip the source portal to it as well. This will allow us to both reduce the size of the temporary target portal during each recursion as well as reduce the size of the temporary source portal. The smaller the source and target portals become during each recursive step, the smaller our anti-penumbras will become. This means we will clip away many more unseen portals and leaves than would otherwise be the case.

Earlier in the code we clipped the generator portal to the source portal so that if the generator portal was spanning the source portal's plane, we made sure that the generator was split. This left us with only the section of the generator portal that the source portal could see. Now we are going to do the same in reverse (see Figure 17.62).

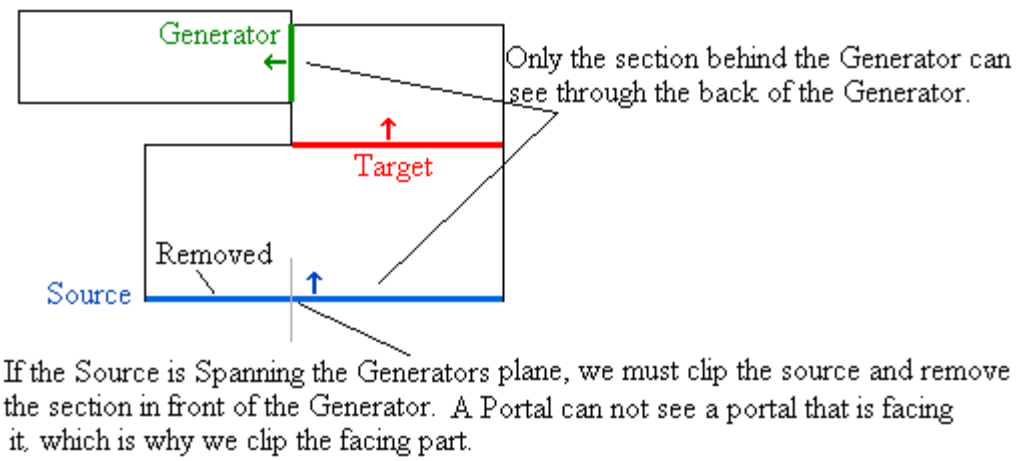


Figure 17.62

Remember that our source portal's PossibleVis array will include any portals that are spanning another portal's plane. In Figure 17.62, we can see that the source portal is indeed spanning the generator portal's plane. As we already know, the only section of the source portal that should be able to see through the generator portal is the section of the source portal that is behind the generator portal. In this case, some of the source portal is in front of the generator portal and is 'facing' the portal. Since facing portals cannot see each other, we have to remove the section of the source portal that is in front of the generator.

The problem is that our Clip function (as is standard for all clippers) clips away the section of the polygon *behind* the passed plane, but we want to do the opposite in this case. That is, we want to keep the piece of the source portal behind the plane and lose the piece in front of the generator plane. This is why we created a reversed generator plane earlier. Instead of having to write another function to remove front pieces, we send in the generator plane with a reversed normal so that the correct section of the source portal gets removed using the same clipper function. This lets the CPortalPoints::Clip function act as if it is removing the back section of the source portal when we are actually removing the front section due to the inverted plane normal.

Here is the code that clips the source portal to the reversed generator plane.

```
        // Make a copy of the source portals points
        SourcePoints = new CPortalPoints( PrevData.SourcePoints, true );

        // Clip the source portal
        NewPoints = SourcePoints->Clip( ReverseGenPlane, false );
        if ( NewPoints != SourcePoints ) FreePortalPoints( SourcePoints );
        SourcePoints = NewPoints;

        // If none remains, continue to the next portal
        if ( !SourcePoints ) { FreePortalPoints(  GeneratorPoints ); continue; }
```

Here we make a copy of the source points and then store them in a local variable called SourcePoints. It is important that we do not alter the actual (master) source points that were passed into this function

from the previous recursion because we are actually in the middle of a loop at the moment. We are looping through each generator portal in the current leaf, so we need to make sure that the original source portal points are left unharmed so that they can be used in the next iteration of the loop. Remember that we will need these points for testing against other generator portals also. The 'true' parameter to the CPortalPoints copy constructor tells it that it should copy that passed source vertices and not just store a pointer to the master vertex set.

If none of the source portal remains after the clipping operation, then the source portal had no points behind the generator plane and we cannot possibly see through it. Therefore, we skip this portal and move on to the next generator in the current leaf.

If all of our portals have survived up to this point then it means that we have rejected any trivial and obvious cases and must now build our anti-penumbra to clip the generator portal so that we are left with only the portion of the generator portal that can be seen through the source and target combination of portals. Fortunately, we are nearly at the end of the code for this function, and all we have to do now is call the ClipToAntiPenumbra function to clip the portals. The ClipToAntiPenumbra function is responsible for actually building the anti-penumbra, as well as clipping the generator portal to it.

In this next section of code, we use a global variable called CLIP_TEST_COUNT to specify how many times we want to call the ClipToAntiPenumbra function. This allows us to set clipping levels between 0 and 3. A setting of 3 will perform four calls to the function and will take longer to compile, but it will create the most accurate PVS. You can set it to any of the other values during level development to speed up the compile process. You will probably only need to set it to the highest level when doing the final compile of your scene, so by default we have set it to level 2.

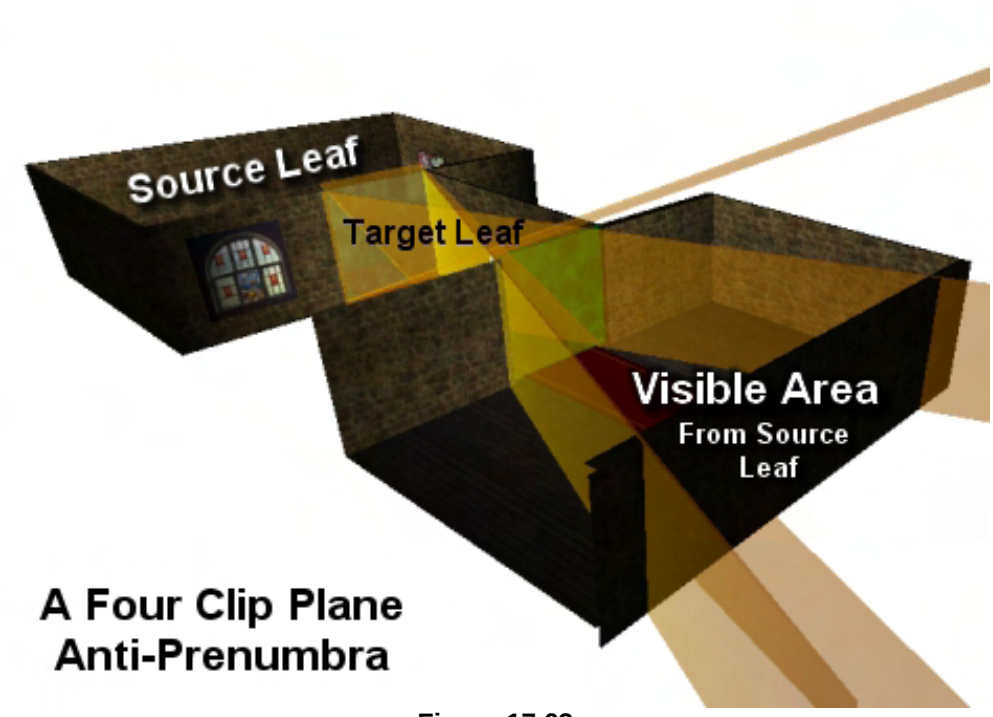


Figure 17.63

For now, just remember that the anti-penumbra is basically just a list of clip planes. Each plane in the list separates the source and target portals so that they both lay on opposite sides of the plane. Figure 17.63 provides a quick refresher of the anti-penumbra theory we discussed earlier. We will talk more about how to actually build the anti-penumbra when we discuss the ClipToAntiPenumbra function a bit later in this lesson.

Next we will look at how we call ClipToAntiPenumbra from the RecursePVS function. At each stage, we test the global variable CLIP_TEST_COUNT to see if it is set high enough to perform each clipping call.

```
        // Lets go Clipping :)
        if ( CLIP_TEST_COUNT > 0 )
        {
            GeneratorPoints = ClipToAntiPenumbra( SourcePoints,
                                                  PrevData.TargetPoints,
                                                  GeneratorPoints, false );
            if (!GeneratorPoints) { FreePortalPoints( SourcePoints ); continue; }

        } // End if 1 Clip Test
```

The first call we make (if CLIP_TEST_COUNT is set higher than 0 -- which it always should be) passes in the source portal, the target portal, and the generator portal. ClipToAntiPenumbra will build an anti-penumbra from the first portal (SourcePoints) to the second portal (TargetPoints) and will clip the third portal (GeneratorPoints) to that anti-penumbra. If the function call returns NULL, then the generator portal was fully outside of the anti-penumbra and cannot be seen from the source portal. If this is the case, then our work is done and we can ignore this portal and move on to the next one in the leaf.

If CLIP_TEST_COUNT is set higher than 1, then we call the ClipToAntiPenumbra function again, switching the order of the SourcePoints and TargetPoints arguments. This builds an anti-penumbra from the target portal to the source portal and clips the generator portal once again. While it would seem at first that building the anti-penumbra from source to target would generate exactly the same clip planes as generating it from target to source, we saw earlier that this would only be true if all portals were of the same shape and size (which is rarely the case).

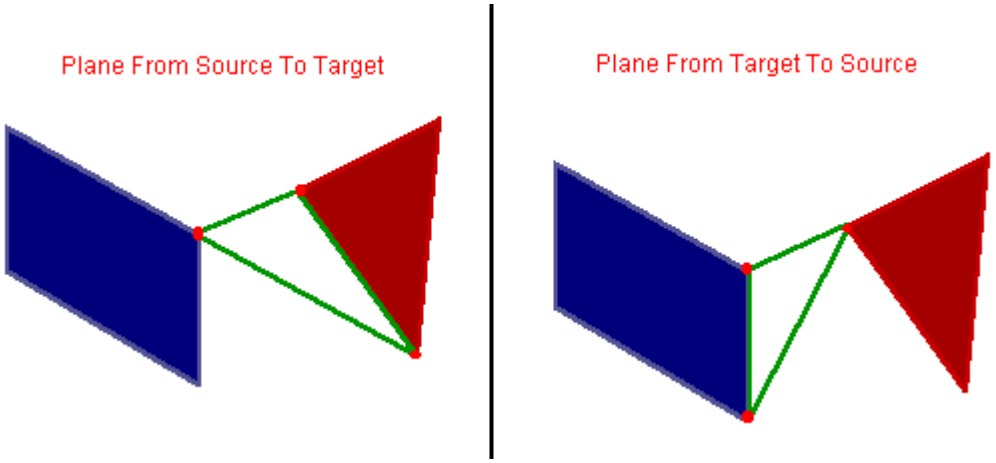


Figure 17.64

Figure 17.64 depicts a square source portal and a triangular target portal. We can see that the clip planes generated by switching the source/target order are quite different, and one combination might allow us to shave a little more off the generator portal than the other. By building the anti-penumbra and clipping to it using both combinations, we make sure that we find the most narrow clip planes possible. It is true that in some cases, both combinations might build the exactly the same anti-penumbra and thus the

second call to ClipToAntiPenumbra would be redundant. However, it is well worth it to include this concept, because in many cases a generator portal might be completely clipped away by the second call, which may not have been the case had we just relied on the first call. Not only will this give us a more accurate PVS, but it will also allow us to prevent further recursion through that portal, which will speed up our calculations. Next we see the second clipping call:

```
        if ( CLIP_TEST_COUNT > 1 )
        {
            GeneratorPoints = ClipToAntiPenumbra( PrevData.TargetPoints,
                                                  SourcePoints,
                                                  GeneratorPoints,
                                                  true );
            if (!GeneratorPoints) { FreePortalPoints( SourcePoints ); continue; }

        } // End if 2 Clip Tests
```

The fourth parameter (the Boolean) to the ClipToAntiPenumbra function requires a quick explanation. When we build our anti-penumbra from the source portal to the target portal, the planes will be created so that they are all facing inwards. In other words, anything outside the anti-penumbra (behind the clip planes) should be clipped away. When we change the order of the source and target portals in the second call however, the winding order will be different and the anti-penumbra is created with outward-facing normals instead. Anything that was behind the planes is now inside the anti-penumbra rather than outside. This means that we will need to toggle the clipping logic in the ClipToAntiPenumbra function so that in this case we keep what is behind and lose what is in front of the clip planes. The Boolean variable tells the function to reverse the clipping process to handle this case. Figure 17.65 depicts this process with the reversed plane normals. On the left we see the anti-penumbra planes being constructed from Source to Target, which generates a plane which, after the crossover in the target leaf, forms an inward facing view volume in the generator leaf. We can see then that any generator portal fragments that lay behind one of these planes get clipped away. When we build an anti-penumbra from target portal to source portal, the normals face into the opposing halfspace and we reject any generator portal fragments that lay in front of a clip plane.

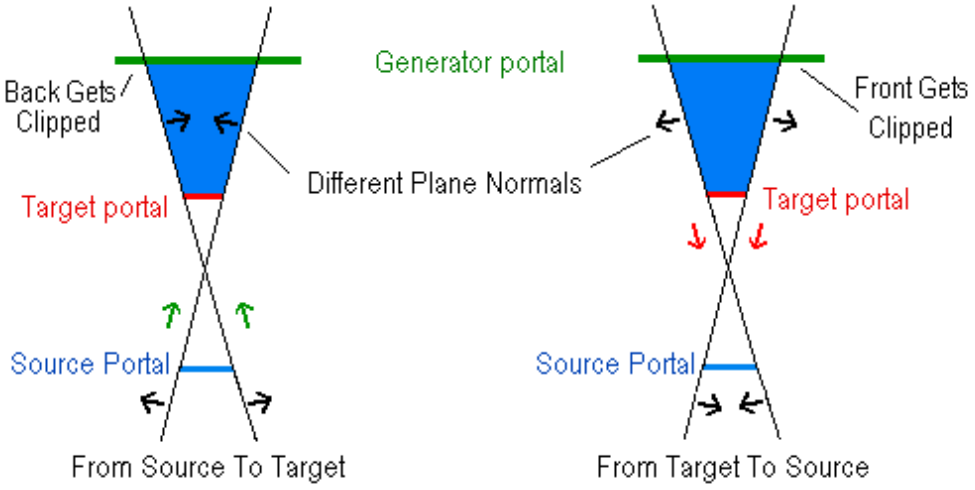


Figure 17.65

The third and fourth calls to the ClipToAntiPenumbra function do virtually the same thing as the first two, only this time they create the anti-penumbra using the target generator portal combination and clip the source portal to it. This means that our source portal will also get smaller with each recursion, resulting in the anti-penumbra that is built each time getting smaller and smaller. Again, this greatly increases our chances of removing more generator portals further down the line.

```
        if ( CLIP_TEST_COUNT > 2 )
        {
            SourcePoints = ClipToAntiPenumbra( GeneratorPoints,
                                               PrevData.TargetPoints,
                                               SourcePoints, false );
            if (!SourcePoints) { FreePortalPoints( GeneratorPoints ); continue; }

        } // End if 3 Clip Test
```

```
        if ( CLIP_TEST_COUNT > 3 )
        {
            SourcePoints = ClipToAntiPenumbra( PrevData.TargetPoints,
                                               GeneratorPoints,
                                                SourcePoints, true );
            if (!SourcePoints) { FreePortalPoints( GeneratorPoints ); continue; }

        } // End if 4 Clip Test
```

So we have actually built four anti-penumbras: two were used to clip the generator portal to the source portal's visibility volume and two were used to clip the source portal to the generator portal's visibility volume. If at any point the source or generator points returned from the ClipToAntiPenumbra call equal NULL, then we know that the portal was completely outside the anti-penumbra, and as a result, the source portal cannot possibly see the generator portal. In this case, we can abort and avoid stepping through this portal.

With all of the clipping now done, reaching this point in the code means that we still must be able to see the generator portal from the source portal through the current target portal. Thus, we must step through that portal and carry on our recursion into the generator portal's neighbour leaf. We store the newly clipped source and generator points in Data so that they can be passed on to the next recursion of this function. Remember that this generator portal will become the next recursion's target portal.

```
        // Store data for next recursion
        Data.SourcePoints = SourcePoints;
        Data.TargetPoints = GeneratorPoints;

        // Flow through it for real
        RecursePVS( GeneratorPortal->NeighbourLeaf, SourcePortal, Data );

        // Clean up
        FreePortalPoints( SourcePoints );
        FreePortalPoints( GeneratorPoints );

    } // Next Portal

    // Clean up
    if (Data.VisBits) delete []Data.VisBits;
```

```
    // Success
    return BC_OK;
}
```

We have now reached the end of this function. Notice that when the function calls itself, it passes in the generator portal's NeighborLeaf so that this leaf will become the current leaf in the next recursion. This means that we are stepping though this portal into that leaf to test the generator portals in that leaf. Our current generator portal is stored in the PVSData structure and passed on to the next recursion so that it can be used there as the new target portal. Finally, we free up any memory we may have allocated for this recursion and exit.

At this point, the hard part is essentially over. We will now examine some placeholder code for a ClipToAntiPenumbra function, which obviously played a pivotal role in the previously discussed function.

17.7.6 The ClipToAntiPenumbra Function

The job of this function is to loop through each edge in the source portal passed into the function and construct a plane using each point of the target portal, and then test whether this plane divides the two portals. We will be looking for the case where the source portal lies on one side of the plane and the target is located on the other. This indicates we have found one of the separating planes we are looking for. If the plane does indeed separate the two portals into two different sub-spaces, then this is considered a valid anti-penumbra clip plane and should be used to clip the generator portal. If the plane does not divide the two portals, then it is ignored

Figure 17.66 depicts an example of both a 'non-separating' and a 'separating' plane. Separating planes are the ones we are trying to find since they describe the extents of the view volume in the generator leaf.

Figure 17.66

Let us now examine the code for this function a few lines at a time:

```
CPortalPoints * CProcessPVS::ClipToAntiPenumbra( CPortalPoints * Source,
                                                 CPortalPoints * Target,
                                                 CPortalPoints * Generator,
                                                 bool ReverseClip )
{
    CPlane          Plane;
    CVector3        v1, v2;
    float           Length;
    ULONG           Counts[3];
    ULONG           i, j, k, l;
    bool            ReverseTest;
    CPortalPoints  *NewPoints;

    // Check all combinations
    for ( i = 0; i < Source->VertexCount; i++ )
    {
        // Build first edge
        l = ( i + 1 ) % Source->VertexCount;
        v1 = Source->Vertices[l] - Source->Vertices[i];
```

We start a loop to iterate through each edge (two vertices) in the source portal using v[i] and v[l] as the two vertices each time. ‘l’ is set up so that it loops back around to zero because the last edge will be between the last vertex and first vertex in the source portal. We use subtraction on these vertices to find the edge vector and store it in vector v1.

To create a valid plane we need to take one point from the target portal and create a vector (v2) from one of the source vertices to the target vertex. This is done for every vertex, creating a different plane during each iteration of the loop.

```
        // Find a vertex belonging to the generator that makes a plane
        // which puts all of the vertices of the target on the front side
        // and all of the vertices of the source on the back side
        for ( j = 0; j < Target->VertexCount; j++ )
        {
            // Build second edge
            v2 = Target->Vertices[ j ] - Source->Vertices[ i ];
```

By performing the cross product on these two vectors we will get the normal of the plane that the edge vectors lie on. If the initial length of this normal is equal to zero (with tolerance) then these two vectors do not lie on or make a valid plane and we should move on to the next point in the target portal.

```
            Plane.Normal = v1.Cross( v2 );

            // If points don't make a valid plane, skip it
            Length = Plane.Normal.x * Plane.Normal.x +
                     Plane.Normal.y * Plane.Normal.y +
                     Plane.Normal.z * Plane.Normal.z;
            if ( Length < 0.1f ) continue;
```

If the plane is valid then we will need to test that it does separate the source and target portals. Before we can use this plane however, we must normalize the plane normal and setup the plane structure.

```
            // Normalize the plane normal
            Length = 1 / sqrtf( Length );
            Plane.Normal *= Length;

            // Calculate the plane distance
            Plane.Distance = Target->Vertices[ j ].Dot( Plane.Normal );
```

Notice how we calculate the plane distance by taking the dot product of any vertex known to be on the plane (we use the target vertex) and the plane normal. We now have a valid plane in the variable called plane (containing a unit normal and a distance to the origin). Now we must see what side of the plane the other points in the source portal are on.

```
            // Find out which side of the  separating plane has the source portal
            ReverseTest = false;
            for ( k = 0; k < Source->VertexCount; k++ )
            {
                // Skip if it matches other verts
                if ( k == i || k == l ) continue;

                // Classify the point
                CLASSIFY Location = Plane.ClassifyPoint( Source->Vertices[ k ] );
                if ( Location == CP_BACK )
                {
                    // Source is on the negative side, so we want all pass
                    // and target on the positive side.
                    ReverseTest = false;
                    break;

                } // End If Behind
                else if ( Location == CP_FRONT )
                {
                    // Source is on the positive sode, so we want all pass
                    // and target on the negative side.
                    ReverseTest = true;
                    break;

                } // End if In Front

            } // Next Source Vertex
```

In the code above, we loop through each vertex in the source portal and classify it against the plane. We need to skip past the two source vertices (i & l), because these were used to create the plane (so they must lie on the plane). As soon as we find a vertex that is either to the front or to the back of the plane then we know that the whole portal must be in front or behind the plane and we can break from the loop. The loop just provides a way to skip past any co-planar vertices. Remember that we will need to remove any piece of the generator portal that is on the same side as the source portal (see Figure 17.67).

Because our clip function will remove backspace fragments, we must set the ReverseTest flag to TRUE if the source portal is in front of the current plane. This is because we want to clip away any areas of the generator portal that are in front of the plane (the same side as source portal). Since our clip function clips away anything that is behind the plane, by setting ReverseTest flag to TRUE, we can flip the normal of the clip plane prior to clipping the generator portal.

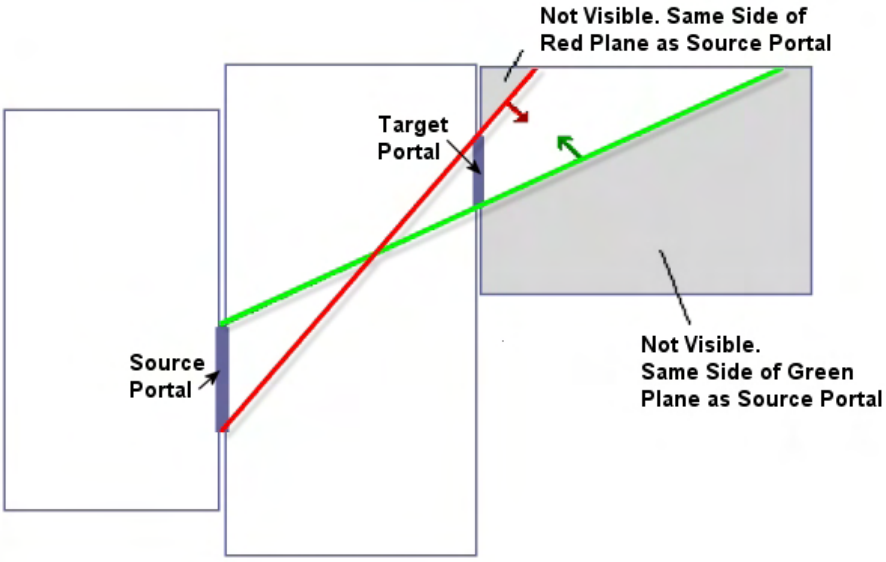


Figure 17.67

```
            // Planar with the source portal ?
            if ( k == Source->VertexCount ) continue;

            // Flip the normal if the source portal is backwards
            if ( ReverseTest ) { Plane.Normal = -Plane.Normal;
                                 Plane.Distance = -Plane.Distance; }
```

If we do not break from the above loop prematurely, it must mean that all of the points are co-planar with the computed plane and that it is not a separating plane. We can see that if k equals Source->VertexCount, then we skip this plane because it is not part of the anti-penumbra. k will only equal VertexCount if the loop is allowed to finish. Next, if the ReverseTest flag was set to TRUE in the loop, then we flip the direction of the plane normal so that the source now lies on the back of the plane instead of the front.

Now that we have a potential dividing plane and have made sure that the source portal is on the back of that plane, we can test the target portal to see if it is on the front side of the plane. If it is, then it is a separating plane and should be used to clip the generator portal. We loop through each vertex of the target portal (excluding the vertex that was used to create the plane) and classify it against the plane. If at any point we find a vertex that is behind the plane, then we bail out, because this cannot possibly be a dividing plane as some of the target portals points are on the same side as the source portal.

```
            // If all of the pass portal points are now on the positive
            // side then this is the separating plane.
            ZeroMemory( Counts, 3 * sizeof(ULONG) );
            for ( k = 0; k < Target->VertexCount; k++ )
            {
                // Skip if the two match
                if ( k == j ) continue;

                // Classify the point
```

```
                CLASSIFY Location = Plane.ClassifyPoint( Target->Vertices[ k ] );
                if ( Location == CP_BACK )
                    break;
                else if ( Location == CP_FRONT )
                    Counts[0]++;
                else
                    Counts[2]++;

            } // Next Target Vertex

            // Points on the negative side ?
            if ( k != Target->VertexCount ) continue;

            // Planar with separating plane ?
            if ( Counts[0] == 0 ) continue;
```

When we get out of the loop, we check to see if the loop was allowed to run its full cycle. If not, it means that we left the loop early because we found a target point that was behind the plane. If this is the case and k is not equal to the target portal's number of vertices, we just skip this plane and move on, because it is not a dividing plane.

Also, if counts[0] == 0 (which was used to record the number of points that were in front of the plane) then every point must be on the plane and again, this is not a dividing plane.

Assuming we pass the prior tests and the code continues, then this is indeed a plane that we want to use in our anti-penumbra and we clip the generator to this plane. You may remember however, that in the RecursePVS call we sometimes passed in TRUE as the final parameter when we needed to reverse the clipping operation and keep the back bit of the generator portal and discard the front bit. In our current situation we likewise must check if this parameter is TRUE, and if so, flip the normal of the plane prior to the call. That is, if the ReverseClip parameter was set to true, the caller would like to remove any portion of the generator portal that is in front of a clip plane instead of behind it.

```
            // Flip the normal if we want the back side
            if ( ReverseClip ) { Plane.Normal = -Plane.Normal;
                                 Plane.Distance = -Plane.Distance; }

            // Clip the target by the separating plane
            NewPoints = Generator->Clip( Plane, false );

            if ( NewPoints != Generator ) FreePortalPoints( Generator );
            Generator = NewPoints;

            // Target is not visible ?
            if (!Generator) return NULL;

        } // Next Target Vertex

    } // Next Source Vertex

    // Success!!
    return Generator;
}
```

This is no different than many of the other calls we have made to clip portals. If the Clip call returns NULL, then the generator portal was completely outside the anti-penumbra and nothing survived. Otherwise, it returns the new clipped vertex list for the generator portal back to the RecursePVS function.

> **Note:** The functions discussed so far in this lesson can be complicated on first read, but they are important to understand. Please be sure to review them a few times if you are having difficulty.

17.7.7 The ExportPVS Function

After the CalcPortalVis function returns program flow back to the ProcessPVS function, every portal will have had its own PVS calculated and stored in its ActualVis array. Recall that this array describes every leaf that the portal can see. However, the PVS rendering code we discussed earlier is not really interested in what the portals can see, it wants to know what a leaf can see (because that will tell us what camera in that leaf can see). The ExportPVS function will handle this conversion for us, using the portals' ActualVis arrays to build a visibility set for each leaf. Once done, all of the leaves' visibility information will be zero run length compressed and stored in one master array inside the tree. This array will represent the final Potential Visibility Set that we save to disk and eventually use in our rendering component.

As it turns out, this function will be very small since we have already done all of the hard work. All we must do is answer the question, "How do we know which leaves a given leaf can see?" Of course, we already know the answer to that: a leaf can see everything that all of its portals can see. So to get a complete visibility set for each leaf, we will just iterate through every portal owned by that leaf and concatenate all of the bits in their ActualVis arrays into one final result.

A temporary buffer (called LeafPVS) will be used to collect and store the bits for each leaf as we go along. This leaves us with an uncompressed buffer much like the ActualVis array that the portals use -- where one bit per leaf is used to indicate visibility. This buffer has the bits set for the leaves that all of its portals can see and thus we have a complete visibility set for that leaf. The next step is to take this buffer and send it to the CompressLeafSet function to compress the bits using zero run length encoding. Then we can insert the results into the master PVS data array stored either in the tree or sent out to file. Remember that every leaf will have its compressed PVS set stored in this one large array that the leaves of the BSP tree can index into. Earlier you will recall that we added a PVS member to the BSP tree's leaf structure. Every leaf will have a PVS index member that specifies where in the master PVS data block its PVS set starts. This index will also need to be set in this function.

```
HRESULT CProcessPVS::ExportPVS( CBSPTree * pTree )
{
    UCHAR * PVSData = NULL;
    UCHAR * LeafPVS = NULL;
    ULONG   PVSWritePtr = 0, i, p, j;


        // Reserve Enough Space to hold every leafs PVS set (plus some to spare)
        PVSData = new UCHAR[(pTree->GetLeafCount() * m_PVSBytesPerSet];
```

```
        // Set all visibility initially to off
        ZeroMemory( PVSData, pTree->GetLeafCount() * m_PVSBytesPerSet);

        // Allocate enough memory for a single leaf set
        LeafPVS = new UCHAR[ m_PVSBytesPerSet ];
```

At the head of the function we allocate an array where the master PVS data block will be stored. This array should be large enough to store N bytes for each leaf in the tree, where N is the number of bytes needed to store 1 bit for every leaf in the tree. In other words, this array has to be large enough to contain the visibility information of every leaf, for every leaf. We will hopefully not need anywhere near this much room to store that data since it will be compressed as it is added, but we can always shrink this array later when we know the final size of the compressed data block. We also allocate the temporary buffer that will be used to hold the leaf's uncompressed visibility set. We will reuse this same buffer for each leaf which is why it is allocated to be large enough to hold a bit for every leaf in the tree.

The PVSWritePtr variable is very important because we will use it to keep track of where we want the compressed data to be written in the master array. We will set it to 0 initially because we want the first leaf's PVS to be compressed into the master array starting at the beginning. We will see how this works in a moment. Now we are ready to loop through every leaf in the global BSP tree's leaf array and collect the visibility bits from each of its contained portals.

```
        // Loop round each leaf and collect the vis info
        // this is all OR'd together and ZRLE compressed
        // Then finally stored in the master array
        for ( i = 0; i < pTree->GetLeafCount(); i++ )
        {
            CBSPLeaf * pLeaf = pTree->GetLeaf(i);

            // Clear Temp PVS Array Buffer
            ZeroMemory( LeafPVS, m_PVSBytesPerSet );
            pLeaf->PVSIndex = PVSWritePtr;
```

For each leaf, we first clear the temporary buffer (LeafPVS) and set the current leaf's PVS index member equal to PVSWritePtr. This tells the leaf where its compressed PVS set will begin in the master PVSData array so that we can access its bits when rendering the tree. As we write each leaf's buffer to the array in compressed format, this variable will be updated to point to the position where the next leaf should be written when being compressed. For the first leaf, we already know that this will be at position 0.

Note that a leaf will always be visible to itself, so we set this leaf's bit in the temporary buffer to 1 by calling the SetPVSBit function.

```
            // Current leaf is always visible
            SetPVSBit( LeafPVS, i );
```

Now we will loop through every portal in the leaf and do a bitwise OR between the temporary buffer (LeafPVS) and the current portal's ActualVis array. This makes sure that all bits set to 1 in the ActualVis array in every portal are also set to 1 in the temporary buffer. You should understand the calculation for fetching the one-way portals for the current leaf as we have seen it several times now.

Just remember, the leaves of the tree store portal indices into the original two-way array of portals and we need to convert this into the correct index in the one-way portal array.

```
            // Loop through all portals in this leaf
            for ( p = 0; p < pLeaf->PortalCount; p++ )
            {
                // Find correct portal index (the one IN this leaf)
                ULONG PortalIndex = pLeaf->PortalIndices[ p ] * 2;
                if ( GetPVSPortal( PortalIndex )->NeighbourLeaf == i )
                   PortalIndex++;

                // Or the vis bits together
                for ( j = 0; j < m_PVSBytesPerSet; j++ )
                {
                      LeafPVS[j] |= GetPVSPortal( PortalIndex )->ActualVis[j];

                } // Next PVS Byte

            } // Next Portal
```

Now we will loop through every portal in the leaf and do a bitwise OR between the temporary buffer (LeafPVS) and the current portal's ActualVis array. This makes sure that all bits set to 1 in the ActualVisBits array are also set to 1 in the temporary buffer.

When the above loop finishes we will have collected all of the bits from all of the portals owned by this leaf and we will have a complete visibility set for it. Now we just call the CompressLeafSet function to add it to the master PVSData array.

```
            // Compress the leaf set here and update our master write pointer
            PVSWritePtr += CompressLeafSet( PVSData, LeafPVS, PVSWritePtr );

        } // Next Leaf
```

When we call CompressLeafSet, we pass in the temporary leaf visibility buffer along with the location within the master array where we want the compressed bits to be written (PVSWritePtr). We also pass in as the first parameter the beneficiary of the compressed data. After the function has compressed the bits and written them to the passed master array, the function returns the number of bytes that the leaf PVS buffer was compressed to before it was added to the master array. All we have to do now is add this number to the PVSWritePtr and we will have the updated location to begin writing the next leaf's PVS in our next iteration. This also gives us the correct value for the next leaf's PVSIndex member used during rendering.

When the loop ends, our scene PVS is complete and is stored in the PVSData array allocated at the top of this function. This array is ready to be saved to file and used by the runtime component. Now we can release the temporary buffer memory that we allocated at the start of this function.

```
        // Clean up after ourselves
        delete []LeafPVS;
        LeafPVS = NULL;

        // Pass this data off to the BSP Tree (data, size, compressed)
```

```
        pTree->SetPVSData( PVSData, PVSWritePtr, PVS_COMPRESSDATA ) ;

        // Free our PVS buffer
        delete []PVSData;

    // Success!!
    return BC_OK;
}
```

At the bottom of the function the master PVSData pointer is passed to the BSP tree's SetPVSData method. This method is also passed the compressed size of the data, so it will resize the passed PVSData array (actually done via a copy into a smaller array) and store it in one of its member variables. Other processes such as those that save the tree to file can then access the PVS data.

17.7.8 The CompressLeafSet Function

CompressLeafSet is responsible for taking the information out of the passed leaf visibility buffer, compressing it using zero run length encoding, and adding it to the master PVSData array. You will recall that it is invoked in ExportPVS like so:

```
// Compress the leaf set here and update our master write pointer
PVSWritePtr += CompressLeafSet( PVSData, LeafPVS, PVSWritePtr );
```

The function is passed the leaf visibility buffer and the position to start writing in the master PVSData array. Remember that PVSWritePtr equals 0 the first time this function is called (for leaf 0's visibility set), so we will be telling the function to start writing the compressed information at byte 0 in the PVSData array. CompressLeafSet will return the number of compressed bytes written to the PVSData array during the current call. This value was added to the write pointer in previous function to allow us to skip past all of the compressed data that has just been written to the master PVSData array. The write pointer then points to the offset in the array where we should start writing the next leaf's compressed data the next time this function is called.

```
ULONG CProcessPVS::CompressLeafSet ( UCHAR MasterPVS[],
                                     const UCHAR VisArray[],
                                     ULONG WritePos)
{
    ULONG   RepeatCount;
    UCHAR  *pDest = &MasterPVS[ WritePos ];
    UCHAR  *pDest_p;

    // Set dynamic pointer to start position
    pDest_p = pDest;

    // Loop through and compress the set
      for ( ULONG j = 0; j < m_PVSBytesPerSet; j++ )
    {
        // Store the current 8 leaves
            *pDest_p++ = VisArray[j];
```

```
        // Don't compress if all bits are not zero
            if ( VisArray[j] ) continue;

        // Count the number of 0 bytes
            RepeatCount = 1;
            for ( j++; j < m_PVSBytesPerSet; j++ )
        {
            // Keep counting until byte != 0 or we reach our max repeat count
            if ( VisArray[j] || RepeatCount == 255) break;
            else
              RepeatCount++;

        } // Next Byte

        // Store our repeat count
        *pDest_p++ = RepeatCount;

        // Step back one byte because the outer loop
        // will increment. We are already at the correct pos.
            j--;

    } // Next Byte

    // Return written size
      return pDest_p - pDest;
}
```

First we set up two pointers (pDest and pDest_p) so that they point to the correct byte in the PVSData array to start writing the new information. The pDest pointer is not altered after that since its job is to store this starting point for current leaf set. Our outer loop then iterates over every byte in the leaf visibility buffer. (We will actually use pDest_p to write to the PVSData array.)

If the current byte in VisArray is non-zero, then we simply copy it over into PVSData unaltered. This is because we can only compress zeroes, and thus we skip around to the next iteration of the loop. When we do encounter a zero value, then this is the start of a zero run and we know that the next byte that has to be written into the master array is the actual number of zeroes that will follow. We set the RepeatCount variable to 1 at this point because we know that there is at least one zero in the run that we have collected so far. We then execute an inner loop to iterate through the leaf's visibility bits until we hit a non-zero byte or we have reached a run count of 255 (the maximum value capable of being stored in a single byte). Each time we hit another zero, we increase the counter (RepeatCount) by one, thus keeping track of the current length of the run. After exiting the inner loop, the current run length is recorded and we pass control back to the outer loop.

Notice that we decrement the loop counter j by one before we iterate once more in the outer loop. We do this because j is the non-zero byte that will need to be copied into the PVSData array on the next iteration of the outer loop. If we did not do this, as the outer loop iterates and adds 1 to j, we would skip right past the byte that we need to write.

Conclusion

There is no doubt that this was a challenging chapter. Calculating a Potential Visibility Set is certainly no easy task and it would be quite natural for you to require more than one pass through the material before you begin to feel comfortable with the processes and procedures introduced. One of the things we tried to do in this chapter is include a decent amount of implementation specific concepts beyond just the theory of PVS. The hope is that this will set the stage for your studies when you move on to look at the actual lab project code and work your way through the true implementation in the workbook.

We have now reached the end of this course. It is probably fair to say that for many of you, the topics that we covered over the last few months were a bit more difficult than those that we encountered in Module I of this series. But if you take a step back for a moment and have a look at how far you have come in a relatively short while, you should be very proud of yourself. You now have detailed knowledge of many of the most fundamental processes involved in building a 3D game engine. You have looked at rendering, animation, lighting, collision detection, spatial partitioning, and a host of other core topics. The question you probably have at this point is "where do we go from here?" After all, even having reached the end of Module II, a rather large course to say the least, you have yet to have constructed a real game, or even a real game engine for that matter. In truth, this is a very important point and bears some additional discussion.

Module I provided you with the basic set of beginner-level knowledge that you will need when programming 3D applications. We looked at mathematics, lighting, texturing, cameras, alpha blending, and the like. Module II took your skill set to an intermediate level by introducing more complex topics like BSP trees, collision detection and response systems, skeletal character animation, and so on. You probably noticed that there were only a handful of major underlying themes that ran throughout this entire course, even though the things that we built around them seemed very different at times. Module II was primarily a course on geometry management through the use of hierarchies. While other concepts were floated around the periphery, you will note that this has been a key theme in virtually every lesson beyond the first one. We did not cover fancy lighting techniques or special effects; instead we dug deep into the notion of hierarchies and similar ideas and examined ways that we could speed things up by accounting for spatial relationships between the components of our scene. If you think that at this point we are done with hierarchies, guess again. We will see our old friends again as we move into the final phase of this training program.

With that said, you are probably wondering what Module III has in store for you, since you are quite aware at this point that you have still not reached the end of the road.

Well, for starters, it is important to point out that we have not yet even begun to address the topic of game engine design. You might be a little surprised to hear this (maybe even especially after this final chapter on BSP/PVS), but it is true. All we have really done to date is establish the core knowledge base that we will need in order to begin the process of assembling our final game engine. While you could theoretically write a simple rendering engine (and maybe grow it out into a game with some work) using the concepts we covered up to this point, it is not what we would recommend. After all, there are still many things that we have yet to integrate, even if we do have some very impressive pieces of individual technology in our toolbox. Consider the fact that as of right now, our demos are still using fixed function

vertex lighting, which is not exactly the most attractive lighting model, is it? Or think about the fact that for the most part, we are still using only a single texture per-polygon. As you are probably aware from your exposure to games and the game industry, there are lots of things that still need to be discussed before your game engine can begin to take on more modern appearance (bump maps, environment maps, stencil shadows, shaders, etc.). So we have a ways to go yet.

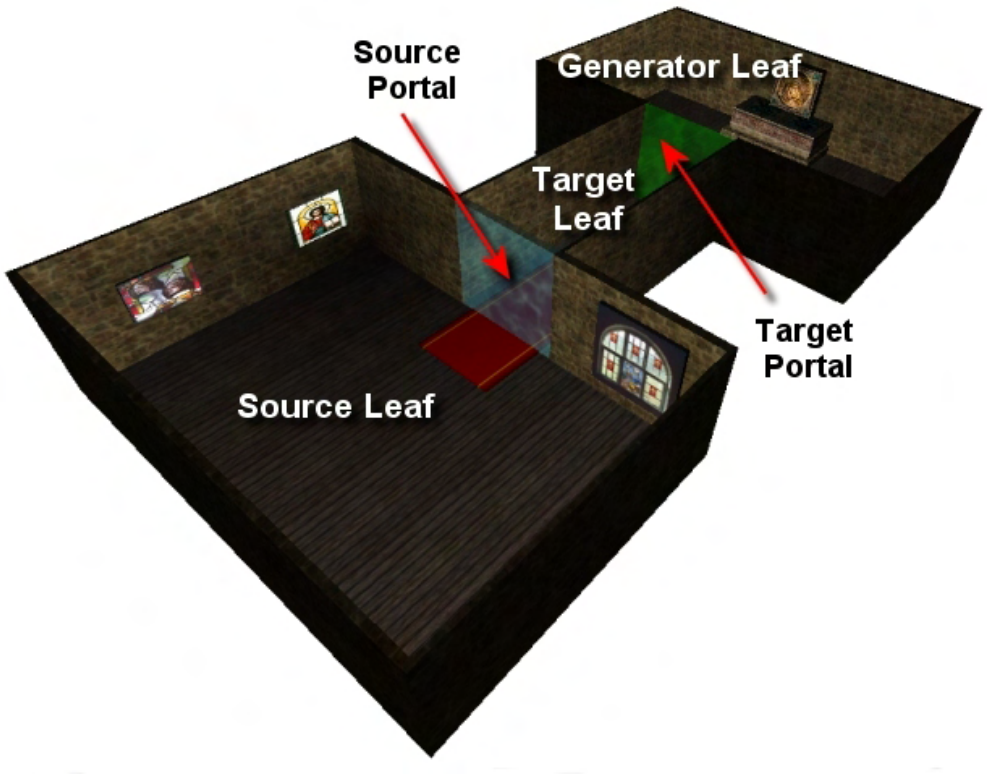
In Module III we will finally begin to wrap up all of these ideas and assemble some very powerful game technology. We will examine the programmable pipeline and introduce the concept of shaders. We will look at various multi-texturing techniques that can significantly improve our visuals. We will talk about various lighting models that provide very realistic results. And of course, we will introduce special effects so that we can include tracers and lasers and explosions and dust trails and the like in our games. And most importantly, we will talk about how all of these components can be properly integrated together into a real commercial-quality game engine. This will involve the introduction of scene graphs, scripting languages, data streaming, and a slew of other vitally important concepts that need to be considered when designing a proper game engine.

Does this mean that we will throw away all of the code that we have written? No. There is plenty of reusable code that will be carried over into future development efforts. What it does mean is that we will be moving away from our very handy but quite limited code framework, whose time has finally come to be put to rest. Actually, we will continue to use it for one-off technical demonstrations when appropriate, but ultimately, a new architecture will need to emerge to take its place.

So those are our goals. Non-trivial goals for sure, but they will introduce some of the most exciting content in the series. We hope that you have enjoyed your studies thus far and that you truly understand the ideas that we have introduced. While it can be a little intimidating at times (game programming is certainly not for the weak and timid), just keep in mind that you have now covered some of the most technically challenging subjects that are on the agenda. Of course, there will be other hard subjects that will come our way, but if you have made it this far, then you should find the series finale a wonderful and rewarding experience.

Good luck to you as you prepare for the final phase. We look forward to completing our journey together and seeing where you will eventually take these ideas on your own.

Workbook Seventeen:
Potential Visibility Sets



Introduction

In this lesson we will continue to add to the lab project implemented in the previous lesson (lab projects 16.2 and 16.3) to facilitate the calculation and rendering of geometry using potential visibility sets. It is imperative that you have read the accompanying text book before continuing with this work book. PVS and portal generation theory will not be discussed in this work book as its intention is to focus on the implementation of these processes into our compiler tool. It is expected that have read the text book and fully understand the theory and placeholder code described there.

In lab project 17.1 we will evolve the development time compiler tool initially implemented in lab project 16.2 by adding two new processing modules to it that collectively make PVS calculation for the input geometry set possible. In addition to the hidden surface removal, BSP leaf tree compile and T-junction repair modules added to our tool in the previous lesson, we will now add a portal generation processing module and a PVS calculation processing module. This will complete the construction of our compiler tool allowing it to compile and save geometry out to an IWF file along with its leaf based PVS data.

Lab project 17.2 will involve a very minor evolution from lab project 16.3 requiring only a few lines of code to be changed in order for it to render geometry using a PVS. As you have no doubt guessed, this lab project (just like 16.3) is being used to represent the run-time component demonstrating how the pre-calculated PVS data can be loaded from file and rendered efficiently by a rendering engine.

Lab Project 17.1: Adding a PVS Calculator to the BSP Compiler

In the first implementation of our BSP compiler tool in the previous chapter, we saw how the compiler itself was constructed as a series of separate processes that were glued together by the CCompiler class. The CCompiler class itself had no idea how to compile a BSP tree, perform HSR or remove T-Junctions from compiled geometry but did know the order in which these processes had to be carried out. The CCompiler class itself had the simple task of loading the geometry from an IWF file and passing this data to the individual processes in a very specified order. The various processes were all implemented in their own classes such as CProcessHSR, CProcessTJR and CBSPTree and as such, as long as CCompiler had knowledge of these objects it could invoke their ::Process methods to instruct them to perform their specific tasks on the data set. With the exception of the application's **main** function having to initially inform the CCompiler object of the IWF file that was to be loaded, all it had to do was issue a call to the CCompiler::CompilerScene function to set this chain of events in motion. When this function returned program flow back to the application's **main** function, the CCompiler class had all the information it needed stored in its compiled BSP leaf tree. The application would finally issue a call to the CCompiler::SaveScene method which would take the information stored in the BSP tree and save it out to file.

The CCompiler::CompileScene function was essentially nothing more than a series of calls to functions that invoked the various processes. We saw that it would first invoke the CProcessHSR module to

perform hidden surface removal on the data set in an attempt to remove hidden surfaces and any illegal geometry that may exist in the data set. After this process returned and had performed its task, the CBSPTree module would be invoked to compile the leaf BSP tree. After the BSP tree had been compiled and program flow had once again returned back to the CCompiler::CompileScene method, the next and final module to be invoked was CProcessTJR which would iterate through the polygons in the compiled BSP tree and would fix any T-Junctions that were found to exist. This would involve T-Junctions that were either introduced via the myriad of clipping operations that were performed via BSP tree construction or that existed in the original loaded dataset.

With the need to now introduce two new processes into the compiler, the CProcessPRT and CProcessPVS modules will be introduced in this lesson and added to the list of processes the compiler must perform. These processes will handle the generation of portals for the data set and the calculation of its leaf based PVS respectively. The CCompiler class will need to be updated to be aware of these new modules and provide a means for storing and passing the compilation options for these modules into the modules themselves.

Because of the modular design of the compiler, the additions to the CCompiler class will be extremely light. The CCompiler::CompileScene method will be updated to invoke additional processes. The order of scene compilation will now be as follows:-

1. Load in the IWF data from file
2. Perform 'Hidden Surface Removal' on loaded geometry to removal illegal surface fragments
3. Compile BSP leaf tree
4. **Generate portals for compiled tree**
5. **Calculate PVS for BSP tree using portals generated in step 4**
6. Perform T-Junction repair on final BSP tree geometry
7. **Construct mesh from BSP geometry and save to file along with PVS and portal Data**

Steps 4 and 5 in the above list are clearly the additions to lab project 16.2 and will be the focus of this work book. These processes will be wrapped in the CProcessPRT (PRT is short for Portal) and CProcessPVS classes respectively.

Step 7 is not actually invoked by the CCompiler::CompileScene method but is activated by the application's main function called the CCompiler::SaveScene method. This function is clearly not a new function and its code is actually unchanged from the previous lesson. However, you will recall that this simple function uses our custom CFileIWF class to perform its IWF file loading and saving. It is the code to this class (in particular its WriteTree method) which has been modified so that it now knows how to write out the additional PVS and portal information that will be stored in the tree.

> **Note:** Although there is no need to save the portals out to file it can sometimes be useful to have them around. You can decide whether you will need them or not and alter the saving code accordingly.

Finally, before we start looking at the code to the new classes and the modifications to previously existing classes, it should be noted that the structures of our CBSPTree class will need to be modified to store both portal information and PVS data. Remember from the text book that all this information needs to be stored in the BSP tree. That is, we need the BSP tree to maintain a list of all the portals that were generated by the portal generation process. Furthermore, we need each of these portals to store in which

leaves they were found to reside. We also need each leaf in the BSP tree (each CBSPLeaf structure) to store an array of portal indices into the tree's portal array describing exactly which portals were found to reside in that leaf. With respect the calculation of the PVS set, the CBSPTree class must also have a place to store the compiled PVS data (the compressed visibility bit set for each leaf in the tree) and each leaf in the tree must be modified to also store a PVS index. The leaf's PVS index will describe the location in the tree's main PVS array where its visibility information will begin. You are reminded that after the PVS has been calculated, the visibility information for every leaf in the tree will be stored in this one master PVS data array which is why each leaf needs to know where within that array its visibility information is located.

We will start our discussion of the lab project 17.1 source code by examining the changes to the top level object that binds everything together, the CCompiler class.

The CCompiler Class - Updated

The CCompiler class has changed very little from its initial conception in lab project 16.2. The complete class declaration is shown below with any new members highlighted in bold. Remembering that the CCompiler class stores a set of compilation options for each process that it invokes, we can see that it now has two new member variables PRTOPTIONS and PVSOPTIONS. These are structures that contain the compilation options for the portal generator module and the PVS calculator module respectively. Notice, how two new methods have also been added called PerformPRT and PerformPVS. These are the methods called from the CCompiler::CompileScene method to invoke the portal generation and PVS calculation modules. This is not a new concept to us as we discussed how each process invoked by the compiler has a matching member function that performs this task (such as PerformHSR and PerformTJR etc).

Excerpt from CCompiler.h

```
class CCompiler
{
public:

    // Constructors & Destructors for This Class.
             CCompiler();
    virtual ~CCompiler();


    // Public Functions for This Class.
    bool            CompileScene     ( );
    void            Release          ( );
    void            SetFile          ( LPCTSTR FileName );
    void            SetOptions       ( UINT Process, const LPVOID Options );
    void            GetOptions       ( UINT Process, LPVOID Options ) const;
    void            SetLogger        ( ILogger * pLogger ) { m_pLogger = pLogger; }
    CBSPTree       *GetBSPTree       ( ) const { return m_pBSPTree;   }
    bool            SaveScene        ( LPCTSTR FileName );

    void            PauseCompiler    ( );
    void            ResumeCompiler   ( );
    void            StopCompiler     ( );
    bool            TestCompilerState( );

    COMPILESTATUS   GetCompileStatus ( ) const { return m_Status; }
```

```
    void            SetCompileStatus ( COMPILESTATUS Status ) { m_Status = Status; }



    // Public Variables for This Class.
    vectorMesh      m_vpMeshList;       // A list of all meshes loaded.
    vectorTexture   m_vpTextureList;    // A list of all textures loaded.
    vectorMaterial  m_vpMaterialList;   // A list of all materials loaded.
    vectorEntity    m_vpEntityList;     // A list of all entities loaded.
    vectorShader    m_vpShaderList;     // A list of all shaders loaded.

private:

    // Private Functions for This Class.
    bool            PerformHSR( );      // Hidden Surface Removal
    bool            PerformBSP( );      // Binary Space Partition Compilation

    bool            PerformPRT( );      // Portal Compilation
    bool            PerformPVS( );      // Potential Visibility Set Compilation

    bool            PerformTJR( );      // T-Junction Repair


    // Private Variables for This Class.
    HSROPTIONS      m_OptionsHSR;       // Hidden Surface Removal Options
    BSPOPTIONS      m_OptionsBSP;       // BSP Compilation Options

    PRTOPTIONS      m_OptionsPRT;       // Portal Compilation Options
    PVSOPTIONS      m_OptionsPVS;       // PVS Compilation Options

    TJROPTIONS      m_OptionsTJR;       // T-Junction Repair Options

    ILogger        *m_pLogger;          // Logging interface used to log progress etc.
    LPTSTR          m_strFileName;      // The file used for compilation
    COMPILESTATUS   m_Status;           // The current status of the compile run
    ULONG           m_CurrentLog;       // Current logging channel for messages.

    CBSPTree       *m_pBSPTree;         // Our compiled BSP Tree.
};
```

We will see later that the portal processing module has no real options that influence the way in which the portals are compiled. Therefore, the PRTOPTIONS structure contains a single Boolean which describes whether the user would like the compiler to perform the portal generation process. The structure is defined in CompilerTypes.h and is shown below.

Excerpt from CompilerTypes.h

```
typedef struct _PRTOPTIONS {            // Portal Compilation Options
    bool            Enabled;            // Process Enabled ?
} PRTOPTIONS;
```

If the Enabled Boolean is set to false in the CCompiler::m_OptionsPRT structure then the compiler will not call the CCompiler::PerformPRT method and the portal generation module will not be invoked.

The PVSOPTIONS structure has three members which will influence the way in which the PVS calculator will generate its final PVS data. The PVSOPTIONS structure is also declared in the file CompilerType.h and is shown below.

Excerpt from CompilerTypes.h

```
typedef struct _PVSOPTIONS {           // PVS Compilation Options
    bool            Enabled;           // Process Enabled ?
    bool            FullCompile;       // Perform Full PVS Compile
    unsigned char   ClipTestCount;     // Number of portal clip tests to perform
} PVSOPTIONS;
```

The ways in which these members influence the PVS calculator module are described below.

bool Enabled;

This member is no stranger to us as the options structure for each process has this member. In the case of this structure, it describes to the compiler whether PVS compilation should be invoked when compiling the data. It is possible that you might just want to compile a regular leaf tree and may not be interested in PVS calculation. When this is the case, this member can be set to false.

> **Note:** If the portal generation module has been disabled then the PVS calculator will not be invoked even if this member has been set to true. The PVS calculator absolutely needs the portal information to have been generated and stored in the BSP tree in order to perform its task. Enabling the PVS process without enabling the portal generation process will simply cause the PVS module to return prematurely as soon as it is invoked. That is, the PVS module will exit immediately when it discovers there are no portals for it to work with.

bool FullCompile;

This is a very handy option to use when you are developing your tool and do not want to wait potentially hours for your level to compile each time you wish to run your code. You will recall from the text book that the first task performed in calculating the PVS is to generate a very course visibility set for each portal based on nothing more than portal orientations to one another. A temporary list is built for each portal describing which portals could possibly have a flow between them. This is then use to perform a flood fill through the leaves of the tree from the portal currently being processed to very quickly retrieve an array of leaf visibility bits for that portal. Any bits set to zero in this 'Possible Visibility Array' represent leaves that can not possibly be seen from the portal in question. This is done for each portal so that prior to the main clipping process beginning, we have a very rough guide as to which leaves may be visible from a given portal. The main clipping process simply refines this leaf visibility information for each portal by performing anti-penumbra clipping and setting other leaves in the 'Possible Visibility Set' to zero as they are found to exist outside any valid anti-penumbra originating at the current source portal. The result at the end of the PVS process is the 'Potential Visibility Array' for each portal. This is a much refined version of the 'Possible Visibility Array' that was initially calculated at the start of the process and will typically contain much fewer leaves in its visibility set.

Having said all this, it is the calculation of the 'Potential Visibility Array' for each portal that takes so much time due to the exceptionally recursive nature of the procedure and the grotesque amount of clipping that will need to be done to the anti-penumbra planes. It is this process that makes sure that we have the tightest potential visibility set possible. However, this process takes hours to complete on complex data sets and you certainly don't always need to have a full PVS compile performed when testing your application or level geometry. Imagine for example that your artist had just added a new asset to the scene and wanted to see it being rendered in-game. The entire scene would have to be

compiled again forcing the development team to wait several hours before the new level was compiled and able to be loaded into the PVS aware game render engine.

Often, at the development time we are not nearly so picky when testing things such as new assets or new light placement within the scene that our game is actually running with the most optimal PVS data set and would gladly trade off the hours of compile time for a PVS data set that compiles in seconds even if the resulting PVS set generated halved the frame rate of the application. After all, you can save the full-blown PVS compile for when the game is about to burned and shipped.

The calculation of the 'Possible Visibility Array' for each portal is compiled just prior to the main clipping process to aid in the speed up of that process. This array can be thought of as a very over generous visibility set for each portal. As mentioned, the only thing taken into consideration when calculating this array for each portal is the orientations of the portals with respect to one another. No occlusion is considered. However, this data is still in PVS format and can be used as the PVS data instead of calculating the main data set via the core clipping procedure. It takes seconds/minutes to compile instead of hours and is ideal for performing a quick PVS build in order to test some other assets in your game engine. Therefore, if this option is not set to true, a full compile will not be performed and instead, the PVS data calculated will contain the leaf visibility information returned from the simple flood fill for each portal. The main anti-penumbra clipping procedure will not have been invoked to further refine this set. If this option is set to true, the full blown calculator will be invoked and a tight potential visibility set will be laboriously calculated.

unsigned char ClipTestCount;
In the text book we examine how when performing the anti-penumbra clipping process, we could further refine the amount of geometry that made it into the anti-penumbra's volume by building it four times. This would allow us to trim the size of the source and generation portals that got passed into the next recursion by making sure we have the tightest clip planes possible.

In the first step the anti-penumbra planes are built from the source portals vertices to the target portal edges and the generator portal is clipped to it. In the second step we would then reverse the order by building an anti-penumbra from each vertex in the target portal to each edge in the source portal. Once again the generator portal would be clipped to these planes. We showed in the accompanying text book that it was entirely possible for tighter planes to be generated from target to source instead of from source to target and vice versa. Therefore, we build an anti-penumbra in both directions to make sure we clip the maximum possible from the generator portal and reduce its size as much as possible for the next recursion. Obviously, if the generator portal does not survive the any of the two clipping stages described above, the portal is considered not to be visible from the current source portal and we do not recur through it into the neighbor leaf.

In the third test we build an anti-penumbra from what is left of the generator portal and the target portal to create an anti-penumbra with which we can clip the source portal too. We also reverse the plane order and clip the source portal to the anti-penumbra planes generated from the target to the generator portal in a 4^{th} clip test. Once again, if at any point the source portal is entirely clipped away it means that the generator portal can not see it and therefore, no line of sight can possibly exist from the source portal through the generator portal. The temporarily clipped source portal is also passed into the next recursion

with the clipped generator portal so that as we step through the process for a given source portal, the source, target and generator portals becomes smaller and smaller allowing us to generate smaller and tighter anti-penumbras in future recursions and hopefully reject more generator leaves from being recurred into and added to the source portal visibility set.

So we have seen that if all these clipping operations are to be performed, we essentially have to build four anti-penumbras and clip four portals to those anti-penumbras for each generator portal that we visit during the core recursive clipping process. As this clipping is obviously a laborious process to perform each time, the number of these clips that we perform can greatly influence the speed of PVS calculation. When generating the commercial data set it is recommended that you activate all four clipping stages to get the tightest possible PVS data set generated. However, during development tests and debug runs you can perform fewer clip tests. This will speed up PVS compile time at the cost of a more generous PVS in the typical case.

While having an option that allows us to toggle the number of clips performed from between 1 to 4 may seem like a trivial speed up option, it is not the case. Each anti-penumbra test performed means first constructing the anti-penumbra. This involves two loops that iterate through each vertex in the source portal and each edge in the target portal. A plane in constructed for each combination and the source and target portals are classified against it to determine if it is a separating plane. Two more loops enter the fray here as we have to test each vertex in each of the portals against the plane to determine its separation status. Only if the plane has the source and target portal in opposite half spaces will the plane be added to the anti-penumbras clip list. So as we can see, just the generation of the anti-penumbra generated four nested loops. Then of course, we have to loop through each anti-penumbra plane clipping the generator portal to each one which can involve memory allocations and de-allocations when the portal gets clipped. That is an awful lot of work to perform four times for each generator portal that we visit when calculating the visibility set for each source portal.

The ClipTestCount option of this structure can be set from 1 to 4 to determine how many of these clips we would like to perform at each generator portal. Performing only one clip will generate the PVS data array much faster but will typically have a larger number of leaves flagged as visible in each leaf’s potential visibility set.

So we have seen that these two new CCompiler member structures contain compilation options for the portal generation process and the PVS process. Although we will set the compile options for each process in the CCompiler constructor, the options for each process can also be set by the application via the CCompiler::SetOptions function. We will look at the modified code to this method in a moment.

Let us now examine the changes to the CCompiler source code staring with the constructor.

Constructor - CCompiler

The constructor of CCompiler is used to set up the default compile options for each of its modules. The compile options for every module can be altered and retrieved by the application once the CCompiler object has been created via its SetOptions and GetOptions methods respectively.

We saw in the previous lab project how the default options were configured for the hidden surface removal, BSP compile and T-Junction repair modules in this constructor. Now we have added code to also set the default compile options for the portal generation and PVS calculator modules.

```
CCompiler::CCompiler()
{
    // Set up default HSR options
    m_OptionsHSR.Enabled            = true;

    // Set up default BSP options
    m_OptionsBSP.Enabled            = true;
    m_OptionsBSP.TreeType           = BSP_TYPE_SPLIT;
    m_OptionsBSP.SplitHeuristic     = 3.0f;
    m_OptionsBSP.SplitterSample     = 60;
    m_OptionsBSP.RemoveBackLeaves   = true;
    m_OptionsBSP.AddBoundingPolys   = false;

    // Set up default Portal Compile Options
    m_OptionsPRT.Enabled            = true;

    // Set up default PVS Options
    m_OptionsPVS.Enabled            = true;
    m_OptionsPVS.FullCompile        = true;
    m_OptionsPVS.ClipTestCount      = 2;

    // Set up default TJR Options
    m_OptionsTJR.Enabled            = true;

    // Reset Vars
    m_strFileName = NULL;
    m_pBSPTree    = NULL;
    m_pLogger     = NULL;
    m_Status      = CS_IDLE;

}
```

As you can see, the hidden surface removal module has a single compiler option which dictates whether it should be invoked or not. By default, all modules are enabled. As mentioned though, the application can change these settings via CCompiler methods prior to calling the CCompiler::CompileScene method.

The default parameters for the BSP tree build are also unchanged. We enable the process and specify that we would like to build a tree in which the resulting polygons are split/clipped to the leaf nodes in which they reside. We set the splitter choosing heuristic to 3.0 so that we weight the reduction of splits as three times more important than tree balance during splitter selection. We also set the splitter sample to 60 so that in order to speed up BSP tree compile, only the first 60 polygons in the list that make it into each node are tested as split plane candidates. We also specify that we would like any geometry that makes it into back leaves (such as illegal geometry fragments causes by floating polygon rounding errors during the clipping process) to be deleted so that accurate solid/empty space information can be maintained. By default, we also specify that we would not like the BSP compiler to seal our level by surrounding it in an inward facing cube prior to being compiled.

The portal generation options structure has only a single Boolean member describing whether or not this process should be enabled. As mentioned, this is set to true because all modules are enabled by default. The same is true for the T Junction removal options structure.

By default we also enable the PVS calculator and set it to perform a full compile (instead of a quick portal flood) to generate its PVS data. We also set the number of clips to perform at each generator portal to 2 by default. The minimum is one and the maximum is four

SetOptions - CCompiler

The CCompiler::SetOptions function is also not a new function but has now been modified to allow the application to set the compilation options of the two new processes. You will recall that this function accepted as its first parameter the numerical ID of the process which is to have its properties set. The numerical IDs are defined in the file Common.h. We have now added two more numerical defines called PROCESS_PRT and PROCESS_PVS which are used to signify the portal generation module and the PVS calculation modules respectively. Shown below is our current list of numerical defines.

Excerpt from Common.h

```
#define PROCESS_HSR         0   // Hidden Surface Removal
#define PROCESS_BSP         2   // Binary Space Partition
#define PROCESS_PRT         3   // Portals
#define PROCESS_PVS         4   // Potential Visibility Set
#define PROCESS_TJR         5   // T-Junction Repair
```

The second parameter to the CCompiler::SetOptions function is a void pointer which the application can use to pass the relevant options structure for the process it wishes to alter the compilation parameters for. Here is the modified version of the function.

```
void CCompiler::SetOptions( UINT Process, const LPVOID Options )
{
    switch (Process)
    {
        case PROCESS_HSR:
            m_OptionsHSR = *((HSROPTIONS*)Options);
            break;


        case PROCESS_BSP:
            m_OptionsBSP = *((BSPOPTIONS*)Options);
            break;

        case PROCESS_PRT:
            m_OptionsPRT = *((PRTOPTIONS*)Options);
            break;

        case PROCESS_PVS:
            m_OptionsPVS = *((PVSOPTIONS*)Options);
            break;

        case PROCESS_TJR:
            m_OptionsTJR = *((TJROPTIONS*)Options);
            break;

    } // End Switch
```

```
}
```

As you can see, two more case statements have been added to cast the void pointer to either a PRTOPTIONS structure or a PVSOPTIONS structure based on the passed process ID. In each case we copy the contents of the passed structure into the relevant member structure to set the compile options for the passed process. For example, the application could set the options for the PVS process to compile only a 1 clip anti-penumbra PVS by using the following code.

```
CCompiler Compiler;
CompilerSetFile("SomeExampleFile.iwf")

PVSOPTIONS pvsOptions;
pvsOptions.Enabled            =     true;
pvsOptions.FullCompile        =     true;
pvsOptions.ClipTestCount      =     1;

Compiler.SetOptions( PROCESS_PVS , (void*) &pvsOptions );

… Set other process options here …

Compiler.CompileScene ();
```

This is just a simple example but shows how an application using the compiler can configure the compile options of the various processes prior to calling the CompileScene method.

GetOptions - CCompiler

As one might expect, the CCompiler class also exposes a member function to allow the application to fetch the compiler options for a given process. The function takes two parameters with the first being the numerical index of the process the application would like to retrieve the compile time options for. The second parameter is a void pointer to the options structure for the applicable process that will be cast to the correct type inside the function and filled with the compiler settings for that process as shown below.

```
void CCompiler::GetOptions( UINT Process, LPVOID Options ) const
{
    switch (Process)
    {
        case PROCESS_HSR:
            *((HSROPTIONS*)Options) = m_OptionsHSR;
            break;



        case PROCESS_BSP:
            *((BSPOPTIONS*)Options) = m_OptionsBSP;
            break;

        case PROCESS_PRT:
            *((PRTOPTIONS*)Options) = m_OptionsPRT;
            break;

        case PROCESS_PVS:
            *((PVSOPTIONS*)Options) = m_OptionsPVS;
```

```
            break;

        case PROCESS_TJR:
            *((TJROPTIONS*)Options) = m_OptionsTJR;
            break;

    } // End Switch
}
```

Once again, this is not a new method to us but has had two new cases added to the switch statement that allow us to retrieve the compiler options for the portal generation and PVS generation modules.

CompileScene - CCompiler

As we discussed in the previous lesson, it is the CCompiler::CompileScene method that glues the various processes together into a geometry compilation pipeline. In the previous lesson, we saw that this method was responsible for instantiating a CFileIWF object and using it to import the IWF information into its internal mesh, material, texture and entity arrays. This data is then copied from the CFileIWF data vectors into the compiler's own vectors where data that is not related to the BSP compile process (such as entities and materials for example) can be stored and then written back out to the resulting compiled IWF file. None of this is new and is shown below.

```
bool CCompiler::CompileScene( )
{
    CFileIWF IWFFile;

    // Validate Data
    if (!m_strFileName) return false;

    // Retrieve the filename portion only of the string
    LPTSTR FileName = NULL;
    FileName = _tcsrchr( m_strFileName, _T('\\') );
    if (!FileName) FileName = _tcsrchr( m_strFileName, _T('/') );
    if (!FileName) FileName = m_strFileName;
    if (FileName[0] == _T('\\') || FileName[0] == _T('/')) FileName++;

    try
    {
        // We Are starting the process
        m_Status = CS_INPROGRESS;

        // Load the specified file
        IWFFile.Load( m_strFileName );

        // Obtain ownership of the objects loaded
        m_vpMeshList     = IWFFile.m_vpMeshList;
        m_vpMaterialList = IWFFile.m_vpMaterialList;
        m_vpTextureList  = IWFFile.m_vpTextureList;
        m_vpEntityList   = IWFFile.m_vpEntityList;
        m_vpShaderList   = IWFFile.m_vpShaderList;

        // Clear out the IWF's object vectors (don't destroy)
        IWFFile.m_vpMeshList.clear();
        IWFFile.m_vpMaterialList.clear();
        IWFFile.m_vpTextureList.clear();
        IWFFile.m_vpEntityList.clear();
        IWFFile.m_vpShaderList.clear();
```

```
        // Write load success
        if ( m_pLogger )
        {
            m_pLogger->LogWrite( LOG_GENERAL,
                                 0,
                                 true,
                                 _T("Successfully loaded geometry from file '%s'"),
                                 FileName );

        } // End if Logger


    } // End try Block

    catch (HRESULT & e)
    {
        // Write Load Failure
        if ( m_pLogger )
        {
            m_pLogger->LogWrite( LOG_GENERAL,
                         LOGF_ERROR,
                         true,
                        _T("Failed to load geometry from file '%s' with error code '0x%x'"),
                        FileName, e );

        } // End if Logger

        IWFFile.ClearObjects();
        Release();
        return false;

    } // End Catch Block
    catch (...)
    {
        // Write Load Failure
        if ( m_pLogger )
        {
            m_pLogger->LogWrite( LOG_GENERAL,
                                 LOGF_ERROR,
                                 true,
                                 _T("Failed to load geometry from file '%s'"),
                                 FileName );

        } // End if Logger

        IWFFile.ClearObjects();
        Release();
        return false;

    } // End Catch Block
```

At this point the CFileIWF file data has all been copied into the compiler's internal vectors so the CFileIWF data objects can be released as they are no longer needed.

Now we will start the actual compilation process by invoking the various modules one at a time and passing the results of each module onto the next in the chain. As we saw in the previous lesson, the hidden surface removal module is invoked first to remove any illegal geometry and then the BSP leaf tree compiler is invoked to compile the BSP tree. In the previous lesson, the next and final step prior to saving the compiled data out to disk was to perform T-Junction repair on the final compiled polygon

data. However, we now have to processes that must be inserted after the BSP compile process and before the T-Junction repair process. That is, after the BSP tree has been compiled we will then invoke the portal generator to generate the portals for the compiled geometry. These portals will be stored in the BSP tree. The next module to be invoked will be the PVS calculator which will use the portals stored in the tree to generate the final compressed PVS for the BSP tree. This final data will also be stored in the BSP tree in the form of a single compressed data array. Each leaf in the tree will also contain a numerical index into this master array describing where in the array its PVS data begins.

Providing a logging object has been assigned to the CCompiler class we first output to the general channel that the compile process is about to begin. The first parameter to the LogWrite method specifies that this message should be output to the general channel and the second parameter describes this message as being a normal status message so that the default ink color is used for the text (only when being used with a GUI logger). The third parameter instructs the logger that this is the start of a new message so that prior to being printed the logger inserts a line feed to move the text output cursor to the start of a new line.

```
    // Write Log Info
    if ( m_pLogger )
    {
        m_pLogger->LogWrite(LOG_GENERAL,
                            0,
                            true,
                            _T("Beginning compilation run \t\t\t- ") );

    } // End if Logger
```

In the next section we step through each possible module that can be performed and test its options structure to test that its **Enabled** Boolean is set to true. If it is then this means the application would like the compiler to perform that process (assuming the compiler has not been placed into a state where the user has cancelled the compile). As you can see, we first call the CCompiler::PerformHSR function which will invoke the hidden surface removal module and instruct it to removal illegal geometry from the compiler's data set. The second process to be performed is the BSP compiler which is activated via a call to the CCompiler::PerformBSP method. Both of these methods were discussed in the previous lesson.

```
    // Start compiling by removing all hidden surfaces
    m_CurrentLog = LOG_HSR;
    if ( m_OptionsHSR.Enabled && m_Status != CS_CANCELLED) PerformHSR();

    // Build the BSP Tree if requested
    m_CurrentLog = LOG_BSP;
    if ( m_OptionsBSP.Enabled && m_Status != CS_CANCELLED) PerformBSP();
```

Out of the final three modules that are activated (shown below), the first two are new to this application and show two new methods of CCompiler which will be used to invoke the portal generation processor and the PVS calculator modules respectively. The third and final process to be activated is the T-Junction repair module via a call to the CCompiler::PerformTJR method.

```
    // Build the portals if requested
```

```
    m_CurrentLog = LOG_PRT;
    if ( m_OptionsPRT.Enabled && m_Status != CS_CANCELLED) PerformPRT();

    // Build the PVS if requested
    m_CurrentLog = LOG_PVS;
    if ( m_OptionsPVS.Enabled && m_Status != CS_CANCELLED) PerformPVS();

    // Repair any T-Juncs if requested
    m_CurrentLog = LOG_TJR;
    if ( m_OptionsTJR.Enabled && m_Status != CS_CANCELLED) PerformTJR();

    // Clean up if required
    if ( m_Status == CS_CANCELLED ) Release();

    // Processing Run Completed
    m_Status = CS_IDLE;

    // Write end of compilation message (We use warning just to make it blue ;)
    if ( m_pLogger ) m_pLogger->LogWrite( LOG_GENERAL,
                                          LOGF_WARNING | LOGF_ITALIC,
                                          false,
                                          _T("Success") );

    // Success!
    return true;
}
```

As you can see, when the T-Junction removal method returns, the level data will have been compiled and all that is left to do is output the success message (providing the compiler was not cancelled mid way through the chain of events).

The two new sections of code are highlighted in bold above and show the calls to the PerformPRT and PerformPVS methods. These are simple methods that invoke the new modules we will develop in this work book. The code to the PerformPRT method is discussed next.

PerformPRT - CCompiler

The CCompiler::PerformPRT method is called by the CCompiler::CompileScene method after the BSP tree has been compiled. Its task is to initialize and invoke the portal generation module. Although we have not yet discussed the code to the CProcessPRT class (the portal generation module), we can see in the following code how the module is initialized and used. You will notice that this module shares the same interface methods as the other modules making the way in which the compiler interacts with each module consistent for the most part. Each module for example has methods that allow us to set its options, set the logging class to be used and to pass a pointer to the parent CCompiler object that is invoking the module. The ProcessPRT module is only used during the portal generation process and once its Process method has returned, the portal information will all be stored in the BSP tree. Therefore, the CProcessPRT object can be instantiated on the stack as shown in the following code.

```
bool CCompiler::PerformPRT()
{
    // One time compile process
    CProcessPRT ProcessPRT;

    // Set our process options
    ProcessPRT.SetOptions( m_OptionsPRT );
```

```
    ProcessPRT.SetLogger( m_pLogger );
    ProcessPRT.SetParent( this );
```

Now that we have allowed the CProcessPRT object to store a pointer to the logger class our application is using, we will now write some information to the logger stating that the portal generation module is about to start. We clear the logger's channel that has been reserved for portal generation output (LOG_PRT) and output some copyright information and information instructing the user that the portal compilation process is about to begin.

```
    // Write Log Information
    if ( m_pLogger )
    {
        m_pLogger->Clear( LOG_PRT );
        m_pLogger->LogWrite( LOG_PRT,
                             LOGF_WARNING | LOGF_BOLD  ,
                             false,
                              _T("\r\nPortal Processor v1.0.0\r\n"));

        m_pLogger->LogWrite( LOG_PRT,
                             LOGF_WARNING | LOGF_ITALIC,
                             false,
                             _T("Copyright © 2005 GameInstitute.com.
                                 All Rights Reserved.\r\n"));

        m_pLogger->LogWrite( LOG_PRT,
                             0,
                             true,
                             _T("Beginning portal compilation process."));

    } // End if Logger Available
```

As the CProcessPRT object will need access to the BSP tree that it is to generate portals for, its 'Process' method accepts a pointer to a BSP tree as its only parameter. It is this function that is the top level function for the entire portal generation process. That is, when the CProcessPRT::Process function returns, every portal will have been created and will be stored in the BSP tree's portal array. The leaves of the BSP tree will also contain portal indices describing which portals they contain and every portal will contain a two element leaf array describing the indices of the leaves in which they reside.

Here is the remainder of the function that calls the 'Process' method to generate the portals and finally outputs a completion method prior to returning.

```
    // Compile the Portal set
    ProcessPRT.Process( m_pBSPTree );

    // Write Log Information
    if ( m_pLogger )
    {
        if ( m_Status != CS_CANCELLED )
            m_pLogger->LogWrite( LOG_PRT,
                                 0,
                                 true,
                                 _T("Portal compilation completed successfully."));
        else
            m_pLogger->LogWrite( LOG_PRT, 0, true, _T("Portal compilation cancelled."));
```

```
    } // End if Logger Available

    // Success!!
    return true;
}
```

If the process was cancelled for some reason during the portal compile then a cancellation message is output to the portal process logging channel instead.

PerformPVS - CCompiler

This method is almost a duplicate of the previously discussed function with the exception that it instantiates and invokes the CProcessPVS module. We will look at the code to the CProcessPVS class later in the lesson but for now just know that its Process method will build the master compressed PVS data array for the PBS tree.

A CPerformPVS object is instantiated on the stack and the PVS options structure is passed into its SetOptions method. We also inform the module of the logging object we are using and pass a pointer to the CCompiler object that is invoking the module. We then clear the PVS logging channel and output copyright information about the PVS module to that channel.

```
bool CCompiler::PerformPVS()
{
    // One time compile process
    CProcessPVS ProcessPVS;

    // Set our processor options
    ProcessPVS.SetOptions( m_OptionsPVS );
    ProcessPVS.SetLogger( m_pLogger );
    ProcessPVS.SetParent( this );

    // Write Log Information
    if ( m_pLogger )
    {
        m_pLogger->Clear( LOG_PVS );
        m_pLogger->LogWrite( LOG_PVS,
                             LOGF_WARNING | LOGF_BOLD,
                             false,
                             _T("\r\nPVS Processor v1.0.0\r\n"));

        m_pLogger->LogWrite( LOG_PVS,
                             LOGF_WARNING | LOGF_ITALIC,
                             false,
                             _T("Copyright © 2005 GameInstitute.com.
                                 All Rights Reserved.\r\n"));

        m_pLogger->LogWrite( LOG_PVS,
                             0,
                             true,
                             _T("Beginning visibility determination process."));

    } // End if Logger Available
```

After informing the user that the PVS calculator is beginning its process, we call the CProcessPVS::Process method to invoke the top level PVS processing method. When this function returns, the passed BSP tree will have had the master PVS data array stored within it and each leaf will contain and index into this master array describing where its visibility information begins.

```
    // Begin the PVS Process
    ProcessPVS.Process( m_pBSPTree );

    // Write Log Information
    if ( m_pLogger )
    {
        if ( m_Status != CS_CANCELLED )
            m_pLogger->LogWrite( LOG_PVS,
                                 0,
                                 true,
                                 _T("Visibility determination completed successfully."));
        else
            m_pLogger->LogWrite( LOG_PVS,
                                   0,
                                   true,
                                   _T("Visibility determination cancelled."));

    } // End if Logger Available

    // Success!!
    return true;

}
```

As you can see, after the CProcessPVS::Process method returns, we test the status of the compiler and output either a completion message or a cancellation message to the PVS logging channel depending on the outcome.

We have now discussed all the code changes to the CCompiler class and as we have seen, the modular design of the compiler itself makes adding future modules extremely easy. Of course, most of this work book will be dedicated to examining the code to these two new modules (CProcessPRT and CProcessPVS) but before we do, we must look at other classes and structures that will need to be changed to accommodate the storage of the information these two modules will provide our tree.

In the next section we will examine the changes to the CPolygon class and will examine a new CPolygon derived class called CBSPortal. The CBSPPortal object will be used by the CProcessPRT and CBSPTree objects to generate and store the portals of the tree respectively. As discussed in the accompanying text book, a portal is simply a polygon with some additional information packaged with it (such as which leaves in the BSP tree they reside in).

The CPolygon Class - Updated

When discussing the generation of portals in the accompanying text book, we learned that the first step in generating a portal is to create a large initial polygon on the plane of the node (currently having a portal generated for it) that is large enough to at least fill the root node’s bounding box along that plane. It was this initial portal that was passed into the BSP tree and clipped to the nodes of the tree so that any

portal fragments that existed in solid space were clipped away. The result of this clipping process was a portal that described the exact shape and size of the ‘doorway’ between two leaves.

Generating an initial portal on the node plane that was large enough to fill the root node’s bounding box was discussed in detail in the text book and requires two inputs, the plane on which the polygon/portal should be constructed an a bounding box describing how large it should be on the plane. As it is pretty useful in other situations to build a polygon on a certain plane that we know will fill some bounding box, we have decided to build this functionality straight into the CPolygon class so that all derived classes (including CBSPPortal) expose it. The modified CPolygon declaration is shown below with the new method highlighted in bold.

Excerpt from CompilerTypes.h

```
class CPolygon
{
public:
        // Constructors & Destructors
                  CPolygon( );
         virtual ~CPolygon( );


        // Public Variables for This Class
        CVertex              *Vertices;             // Polygon vertices
        unsigned long   VertexCount;                // Vertices in this poly


        // Public Functions for This Class
        long             AddVertices( unsigned long nVertexCount = 1 );
        long             InsertVertex( unsigned long nVertexPos );
        void             ReleaseVertices();


        // Public Virtual Functions for This Class
        virtual HRESULT Split( const CPlane3& Plane,
                               CPolygon * FrontSplit,
                               CPolygon * BackSplit,
                               bool bReturnNoSplit = false );

       virtual bool    GenerateFromPlane( const CPlane3& Plane,
                                          const CBounds3& Bounds );

};
```

The GenerateFromPlane method accepts two parameters. The first is the plane on which the polygon should generate its vertices and the second parameter describes a bounding box which the polygon should at least fill (it may be bigger). As the CBSPPortal class used by the portal generator is derived from this class it too will expose this method. This means, the portal generator can simply call this function to generate the initial portal on the node plane by passing in the node that is currently being processed and the bounding box of the root node. This portal can then be passed down the tree and clipped at solid leaves.

GenerateFromPlane - CPolygon

As discussed in the text book, generating a polygon on a plane of a specific size involves first projecting the center of the bounding box onto the node plane to calculate the center of the polygon we are creating. Projecting the center of the bounding box onto the plane is a simple matter of classifying the bounding box center position against the plane to get the distance to the plane from the center point along the plane normal. We can then move the bounding box center point along the plane normal to position it on the plane.

Figure 17.1 shows the center of the bounding box labeled CB and shows how moving it along the plane normal by the distance from CB to the plane creates the point CP. CP is the new center point of our polygon and the point from which the four vertices of the portal quad will be placed relative too.

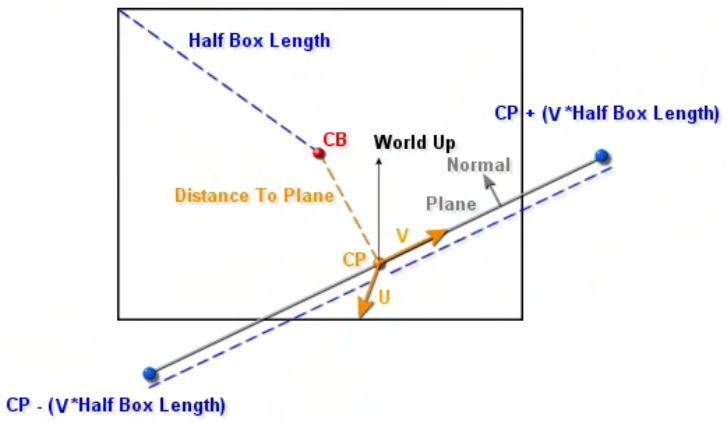


Figure 17.1

After CP has been calculated we have to generate the tangent and bi-normal vectors of the plane shown as vectors U and V in figure 17.1.

Generating the U and V vectors is simple with the cross product at our disposal. We first find any vector which is not identical to the plane normal. This is important because by crossing this vector with the normal we will get vector U, a vector that is perpendicular to the plane normal and thus tangent to the plane. If the vector we choose to cross with the normal is identical to the normal the two input vectors to the cross product will be the same and the resulting vector is undefined.

In figure 17.1 the world up vector is used but any vector can be used as long as it is not the same as the normal vector. In our code we test the world X, Y and Z axis vectors and choose the one that is least aligned with the plane normal. This assures that our cross product will not have any epsilon issues if the two input vectors are nearly equivalent. Once vector U is generated we can simply cross it with the plane normal to get vector V. At this point, these vectors should be unit length.

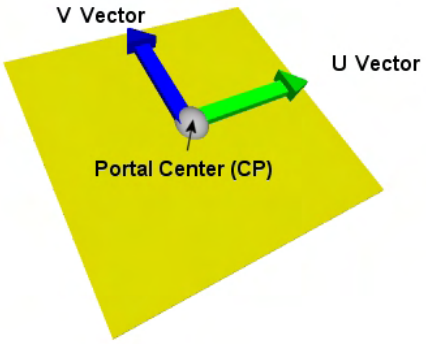


Figure 17.2

Next we get the half length vector of the bounding box by subtracting from the maximum box extent vector, the center position of the box. We then retrieve the length of this resulting half vector. Once we have the length we can use this to scale the U and V tangent vectors so that they can be used in combination with the polygon center point to describe the four corner vertex positions if the polygon. These four vertices are then generated and stored in the polygons vertex array.

> **Note:** It is clear that as this method generates the vertices of the polygon, it should only be called for CPolygon's (and derived objects) which currently contain no vertex data. For example,

```
CPolygon Poly;
Poly.GenerateFromPlane ( SomePlane , SomeBox);
```

The code to this method is fairly short and is shown below. We first calculate the center of the polygon by calculating the distance from the bounding box center point to the plane and then moving the point along the reverse plane normal to locate a point on the plane (CP).

```
bool CPolygon::GenerateFromPlane( const CPlane3& Plane, const CBounds3& Bounds )
{
        CVector3 CB, CP, U, V, A;

        // Calculate BBOX Centre Point
        CB = Bounds.GetCentre();

        // Calculate the Distance from the centre of the bounding box to the plane
        float DistanceToPlane = CB.DistanceToPlane( Plane );

        // Calculate Centre of Plane
        CP = CB + (Plane.Normal * -DistanceToPlane );
```

Next we need to generate the tangent vector and in order to do this we need a vector to cross with the plane normal which must not be identical to the plane normal. The vector A is calculated by analyzing the plane normal components and generating an axis aligned vector that is least aligned to the normal.

```
        // Calculate Major Axis Vector
        A = CVector3(0.0f,0.0f,0.0f);

         if( fabs(Plane.Normal.y) > fabs(Plane.Normal.z) ) {
              if( fabs(Plane.Normal.z)  < fabs(Plane.Normal.x) ) A.z = 1; else A.x = 1;
        } else {
              if (fabs(Plane.Normal.y) <= fabs(Plane.Normal.x) ) A.y = 1; else A.x = 1;
    } // End if
```

We then cross vector A with the plane normal to generate vector U and then cross the normal with vector U to generate vector V. These vectors are then both normalized.

```
        // Generate U and V vectors
        U = A.Cross(Plane.Normal);
        V = U.Cross(Plane.Normal);
        U.Normalize(); V.Normalize();
```

In the next step we calculate the length of the vector from the center of the bounding box to the box corner which describes the furthest the edges of the box could ever be from the polygon. We then scale the unit tangent vectors by this amount.

```
        float Length = (Bounds.Max - CB).Length();

        // Scale the UV Vectors up by half the BBOX Length
        U *= Length; V *= Length;
```

With these two vectors and the polygon center point we can now combine them to generate the position vectors of the four vertices of the quad polygon we wish to create on this plane which are stored in a temporary 3D vector array called P in the following code.

```
        CVector3 P[4];
        P[0] = CP + U - V; // Bottom Right
        P[1] = CP + U + V; // Top Right
        P[2] = CP - U + V; // Top Left
        P[3] = CP - U - V; // Bottom Left
```

With these corner positions computed we then allocate space in the polygon's vertex array to store these four vertices and then copy them into the vertices one by one. The normal of each vertex is assigned the normal of the plane.

```
        // Allocate new vertices
        if (AddVertices( 4 ) < 0) return false;

        // Place vertices in poly
        for ( int i = 0; i < 4; i++)
        {
             Vertices[i] = CVertex(P[i]);
             Vertices[i].Normal = Plane.Normal;
         } // Next vertex

    // Success!
    return true;
}
```

If you are feeling a little rusty on this procedure, please refer back to the accompanying text book where we describe this process in more detail. This completes our coverage of the modified CPolygon object. As you can see, we simply added a new member function.

The CBSPPortal Class

The portal generation module will need a structure with which it can use to represent portal data within the BSP tree. Although a portal is just a polygon (geometrically speaking), we can not use CPolygon to represent them as we need to package additional information with each portal. Such information includes storing the indices of leaves in which the portal is found to reside and the number of leaves in which the portal resides. As discussed in the text book, each valid portal will always reside in exactly two leaves so this leaf count member would be used during portal generation to delete any portal fragments that are found not to exist in two. The portal must also store information about the node in the tree on whose plane it has been created. This is also important information to have during the portal generation process as it allows us to identify the two leaves under that node which are the valid leaves in which the portal should reside. If the portal is found to pop out in a leaf which is above the portals owner node, it is a rogue fragment and can be discarded.

> **Note**: It is vitally important that you have read the accompanying text book before progressing through this work book. Portal and PVS generation is a complex subject and theory of these processes will not be rehashed again in this work book. In short, if you have not studied the text book you find you understand very little of the discussions that follow.

The process that clips the portal to the BSP tree will at some point through the process will have split the initial portal in a list of portal fragments. As these fragments must be grouped together and passed

through the tree also, the portal structure will also have a **NextPortal** member that will allow us to string multiple CBSPPortal structures together in a linked list.

The CBSPPortal class is declared in CBSPTree.h and is shown below. Notice that it is derived from CPolygon so we do not have the reinvent the wheel with respect to its vertex management and member functions. This means the CBSPPortal will also expose the 'CreateFromPlane' member previously discussed which will be used by the portal generator to build the initial portal quad on the node plane.

Excerpt from CBSPTree.h

```
class CBSPPortal : public CPolygon
{
public:

    // Constructors / Destructors for this Class
    CBSPPortal( );

   // Public Variables for This Class
    CBSPPortal     *NextPortal;         // Linked List Next Portal Item
    unsigned char   LeafCount;          // In how many leaves this portal resides
    unsigned long   OwnerNode;          // Node that created this portal
    unsigned long   LeafOwner[2];       // Front / Back Leaf Owner Array

   // Virtual Public Functions for This Class
   virtual HRESULT Split( const CPlane3& Plane,
                          CBSPPortal * FrontSplit,
                          CBSPPortal * BackSplit );

};
```

Notice that we have added four new members to those inherited from CPolygon which allow us to store the indices of the leaves in which the portal is eventually found to reside and the node the node on which the portal was generated. Notice however that we also implement a new Split function that accepts CBSPPortal pointers. Do not worry! We do not have to implement a polygon splitting function all over again. As you will see in a moment, the Split function simple calls the base class's Split function and then copies the extra portal information into the two resulting front and back splits. Let us list those member variables and describe their purpose.

unsigned long OwnerNode;
As we know, all the nodes of the BSP tree are stored in its master node array. We also know that an initial portal will need to be generated on every node that does not have solid space behind it. This portal will then be passed down the tree and any portions of the portal that exist in solid space will be clipped away. This member contains the index of the node in the tree's node array whose plane this portal was created on. This is used during portal generation to make sure that when a portal ends up in a leaf, it is a leaf that is below the owner node in the tree. If this is not the case then this is a rogue fragment and can be deleted.

unsigned long LeafOwner[2];
Every valid portal will always exist in exactly two leaves as a portal by its very nature is a polygon that represents a door way between leaves and as such, has a leaf on either of its sides. Each valid portal will contain the indices of the two leaves in which it resides in this array. Therefore, this array tells us the two leaves for which this portal forms a doorway.

unsigned char LeafCount;
This member is used during portal generation to keep track of how many leaf indices we have currently added to the above array. This member will start of at zero when the initial portal is first fed into the root node of the tree and will be incremented each time the portal pops out in a leaf. This will never be higher that 2 as a portal can only ever possibly exist in two leaves. However, that does not mean that the initial portal will not end up in many valid portals being created which all exist in a different set of leaves throughout the level. Remember, when a split plane is chosen for BSP creation, the entire geometry set beneath that node in the tree is split. That split may have caused multiple convex leaves to be created across the entire level and therefore, therefore, multiple portals may exist on this node plane to bridge the gaps between those pairs of leaves.

CBSPPortal *NextPortal;
As we showed in the text book, the portal/BSP tree clipping process will require keeping track of al the fragments that a portal gets split into via linked lists. This member allows us to storing multiple CBSPPortal fragments together into a linked list so that we can pass the entire list down the tree by simply passing a pointer to the head of the list through the recursive process.

Constructor - CBSPPortal

The constructor of CBSPPortal simply initializes the portals members to their invalid default values. Each element in the LeafOwner array is set to zero and so is the leaf count. The NextPortal member is set to NULL and the owner node index is set to -1.

```
CBSPPortal::CBSPPortal()
{
    // Initialise any class specific items
    LeafOwner[0]    = 0;
    LeafOwner[1]    = 0;
    NextPortal      = NULL;
    LeafCount       = 0;
    OwnerNode       = -1;
}
```

When the portal object is first created we can see that it is not a valid portal. The portal generation process, after creating a new CBSPPortal object will first construct its geometry via a call to its (inherited) GenerateFromPlane method. This will build the polygon on the chosen node plane and will fill the CBSPPortal with geometry representing a large quad on the node plane. This CBSPPortal will then be passed down and clipped to the BSP tree where the above members will eventually get assigned their final values.

Split - CBSPPortal

Every time a portal is split during the portal generation process, we must not only split the geometry into two child portals but must also carry over the information recorded in the parent portal into the two children. It is vitally important that the journey of the parent is inherited by the children before the parent is deleted for the portal generation process to work. For example, if it has already been

determined that the parent portal was found to exist in leaf 10, we know that both the children will also exist in leaf 10 assuming they are not found to exist in solid space further down the tree and are deleted. Therefore, we must make sure that this information is carried over into the split fragments.

As you can see in the code that follows, the CBSPortal::Split method simply wraps a call to the CPolygon::Split method which additional code to copy the portal information over into the two child splits.

```
HRESULT CBSPPortal::Split( const CPlane3& Plane,
                                CBSPPortal * FrontSplit,
                                CBSPPortal * BackSplit)
{
    // Call base class implementation
    HRESULT ErrCode = CPolygon::Split( Plane, FrontSplit, BackSplit );
    if (FAILED(ErrCode)) return ErrCode;

    // Copy remaining values
    if (FrontSplit)
    {
        FrontSplit->LeafCount    = LeafCount;
        FrontSplit->OwnerNode    = OwnerNode;
        FrontSplit->LeafOwner[0] = LeafOwner[0];
        FrontSplit->LeafOwner[1] = LeafOwner[1];

    } // End If

    if (BackSplit)
    {
        BackSplit->LeafCount     = LeafCount;
        BackSplit->OwnerNode     = OwnerNode;
        BackSplit->LeafOwner[0]  = LeafOwner[0];
        BackSplit->LeafOwner[1]  = LeafOwner[1];
    } // End If

    // Success
    return BC_OK;
}
```

Notice that we can pass FrontSplit and BackSplit into the CPolygon::Split method even though they are of type CBSPPortal because CBSPPortal is derived from CPolygon.

The CBSPTree Class - Updated

The CBSPTree class will have to be slightly updated in this lab project so that it can now accommodate the portal and PVS information that it must now also store. The BSP tree will now maintain a vector of all the valid CBSPPortals that were generated via the portal generation process. That is, once the portal generation module has found a valid two leaf portal, it will add its pointer the tree's master portal array.

The CBSPTree class will also need to have some mechanism of storing the final PVS data array that will be generated by the PVS module. As discussed in the text book, the PVS data whether compressed or uncompressed will be represented as a byte array and therefore, the BSP tree will now have an unsigned char pointer that points to this block of data. It will also need a member variable to the store the size of the PVS data block and a Boolean specifying whether the data has been compressed. The compression

technique used is called ‘zero run length encoding’ which compresses runs of zero bytes up to 255 bytes in length into two bytes. This compression technique is discussed in the text book and lecture.

The CBSPTree class is declared in the file, CBSPTree.h. We will not show the whole class declaration here as it is getting rather large and we have only added four new members. Therefore, below we show just the members that have been added in this lab project to facilitate the storage of the portal and PVS data that will be generated by the CProcessPRT and CProcessPVS modules respectively.

Excerpt from CBSPTree.h

```
class CBSPTree
{
public:

    // Public Variables for This Class.
    UCHAR          *m_pPVSData;             // PVS Data set (array)
    unsigned long   m_lPVSDataSize;         // Size of the PVS data set
    bool            m_bPVSCompressed;       // Is the PVS data compressed

private:
    // Private Functions for This Class.
    vectorBSPPortal m_vpPortals;            // Portals built by the CProcessPRT compiler.
};
```

These new member variables are described below.

UCHAR *m_pPVSData;
After the PVS calculation module has generated a PVS for every leaf in the tree and merged them together into a single array, this pointer will be assigned to point at that master PVS data block. This is the PVS data that will be saved out to file and utilized by the run time component.

This member pointer is assigned to the PVS data by the CProcessPVS module via a call to a new CBSPTree member function called SetPVSData. This function will be passed a UCHAR pointer to the PVS data, the size of the PVS data array and a Boolean describing whether the data is in compressed format. We will look at the code to the SetPVSData method in a moment.

unsigned long m_lPVSDataSize;
This member will be assigned its value via the CBSPTree::SetPVSData method which will be invoked by the CProcessPVS module after the PVS has been calculated. It will describe the size of the UCHAR array of PVS data stored in the above member.

bool m_bPVSCompressed;
This member will be assigned a value of true or false by the CBSPTree::SetPVSData method which is invoked from the CProcessPVS module. It describes whether the PVS data array is in compressed (ZRLE) format or whether it has been stored as an uncompressed bit set. Obviously, if compiler options have been set such that the data is not compressed, the run time component will need to know this so that it iterates through the PVS data at render time in the correct manner.

vectorBSPPortal m_vpPortals;
This is an STL vector that will be used to store pointers to all the valid CBSPPortal structures generated by the portal generation module. The CBSPTree interface also exposes a SetPortal method which the

portal generation module can use to store a portal to this vector once it has been validated as being valid two leaf portal. The portals in this array will all be two way portals which will later be used by the CProcessPVS module to clone a set of one way portals for PVS calculation.

Because we now have an array (vector) of portals stored in the BSP tree, methods will need to be added that allow us to reserve space in this vector every time we wish to add new portals to the tree. Methods will also be needed to allow us to retrieve the number of portals in this vector and as mentioned, a method will be added to allow another modules to place portal pointers in this vector. Furthermore, the CBSPTree object will also have a function that a calling module can use to allocate the memory for a new CBSPPortal. Let us have a look at these new methods now which will give us an idea of the functions that will be called by the portal generation module, to allocate a new portal, add it to the tree's portal array and retrieve information about that portal.

IncreasePortalCount - CBSPTree

This method is called by the CProcessPRT module every time it wishes to add space for a new CBSPPortal pointer to the end of the BSP tree's portal array. So that the vector is not being continually resized for every valid portal that we find and add to the tree, we use a resizing threshold which should not be new to you. The array resize threshold is set to 100 by default and is assigned the definition BSP_ARRAY_THRESHOLD. Here is the code to the function that allows us to make sure there is at least enough room at the end of the BSP tree's portal array to add a new portal pointer.

```
bool CBSPTree::IncreasePortalCount()
{
    try
    {
        // Resize the vector if we need to
        if (m_vpPortals.size() >= (m_vpPortals.capacity() - 1))
        {
            m_vpPortals.reserve( m_vpPortals.size() + BSP_ARRAY_THRESHOLD );

        } // End If

        // Push back a NULL pointer (will already be allocated on storage)
        m_vpPortals.push_back( NULL );

    } // Try vector ops

    // Catch Failures
    catch (...)
    {
        return false;

    } // End Catch

    // Success
    return true;
}
```

The first thing we do in the above code is fetch the size of the vector. This tells us how many portal pointers are currently stored in that vector. We compare this against the capacity of the vector which describes how many portals can be stored in that vector before it is considered full. As the purpose of this function is to assure that the capacity is at least 1 greater than the current size so that there is room

to add another portal pointer, a compare between the two is performed. If the size of the vector is greater or equal to the capacity then it means the vector is full and we must resize it. However, instead of simply resizing the vector by 1 to make room for the a new portal, we resize by our threshold value which by default will resize the vector making room for 100 more portal pointers. Why do we do this? Because array resizes are expensive and we do not want to be performing one for every single portal that we add. This way, we make sure that we only cause a resize every 100 portals even if that means at the end of the process we have a little unused capacity in the vector.

You can see that if the capacity is full we reserve more space so that the vector is large enough to contain its current data (size) plus the 100 (BSP_ARRAY_THRESHOLD) new elements. Notice how we push a NULL pointer on the back of the array which forces the size of array to be increased by 1. You will see why this is necessary in a moment as.

GetPortalCount - CBSPTree

This simple function allows a calling module to inquire about how many portals are currently contained in the BSP tree's portal array. It simply returns the size of the vector.

```
unsigned long   GetPortalCount( ) const { return m_vpPortals.size(); }
```

To understand how this might be needed, imagine that we have a CBSPPortal called pMyPortal that we would like to add to the BSP tree's portal array after finding that it is a valid portal. We would first fetch the current portal count of the array as this will also describe the index of the portal we wish to add to the end like so.

```
// Get the current number of portals stored in the tree's array
int PortalIndex = pTree->GetPortalCount();

// The capacity of the array is such that there is enough room to store new portal
pTree->IncreasePortalCount();

// Store the portal at the end of the array
pTree->SetPortal ( PortalIndex , pMyPortal);
```

You will see later that this is exactly the steps taken by the portal generation module each time it wishes to add a new portal to the BSP trees portal array.

AllocBSPPortal - CBSPTree

This function should be used by all modules that wish to allocate a new CBSPPortal. In keeping with our previous strategy we are placing all memory allocation responsibility on the BSP tree object via a series of AllocBSP…method, for all objects that are defined in its header file. This method simply allocates a new CBSPPortal structure and returns it to the caller.

```
CBSPPortal * CBSPTree::AllocBSPPortal( )
{
    CBSPPortal * NewPortal = NULL;
```

```
    try
    {
        // Allocate new portal
        NewPortal = new CBSPPortal;

        // Note : VC++ new does not throw an exception on failure (easily ;)
        if (!NewPortal) throw std::bad_alloc();

    } // End Try

    catch (...) { return NULL; }

    // Success!
    return NewPortal;
}
```

Note that this function does not add the allocated portal to the portal array in any way. It is simply a helper function that wraps the allocation and handles the throwing of an exception if an error occurs.

SetPortal - CBSPTree

When discussing the GetPortalCount method a moment ago, we showed some example code that demonstrated how to add a new portal to the end of the tree's portal array. This protocol involved fetching the current portal count, increasing the size of the portal array by 1 and then setting the portal at the specified position. Here we see the code to the SetPortal function that is used to store the portal pointer in the BSP tree's portal array at the specified position.

The function takes two parameters. The first in the array index where the caller would like the portal to be stored in the array and the second is a pointer to the CBSPPortal structure that is to be stored in the array.

```
void CBSPTree::SetPortal ( unsigned long Index, CBSPPortal * pPortal )
{
  if (Index < m_vpPortals.size()) m_vpPortals[Index] = pPortal;
}
```

Providing that the passed index is within the current size of the array, the value of that array element is replaced with the passed pointer. You might be wondering why the passed index is compared against the size of the vector and not the capacity. To be safe, we only allow the SetPointer method to assign values to elements that are within the current size of the array even if the array has a much larger capacity. As we know that we will always be adding portals to this array one at a time and in order, this just introduces a safety net that would stop us storing portals in non-linear addresses within the array. However, this now clearly demonstrates why in the IncreasePortalCount method we pushed a NULL on the back of the array and forced the size of the array to be increased by one. Were we not to do this we would not be able to use SetPortal to add a new portal to the end of the array. By adding a NULL to the back of the list initially, we create this portal position in the array first and then fill it later when we call the SetPortal method.

GetPortal - CBSPTree

For completeness, whenever there is a Set function there is usually a Get function that performs the reverse operation. The CBSPTree::GetPortal method accepts a single parameter describing the location of the element within the BSP tree's portal array for which the caller would like to fetch the portal pointer. This is fetched from the array/vector and returned to the caller.

```
CBSPPortal* CBSPTree::GetPortal( unsigned long Index ) const
{
   return (Index < m_vpPortals.size()) ? m_vpPortals[Index] : NULL;
}
```

If the passed index is outside the range of the current number of portals stored in the array (<size), NULL is returned.

SetPVSData - CBSPTree

This new member function will be called by the CProcessPVS module to store the compiled PVS data and accompanying information in the BSP tree. The first parameter to this function is where the unsigned char array of PVS data for the entire tree will be passed. The second parameter will describe the number of bytes in this array and the third parameter will describe whether compression was enabled for the CProcessPVS module. This information will be copied and stored in the tree's member variables.

The first thing the function does is delete any PVS data that the tree may already be pointing to with its m_pPVSData pointer as this will now be used to allocate a new block to contain a copy of the passed PVS data. A new byte array is allocated of the correct size (described by the second parameter) and is pointed to by the tree's m_pPVSData pointer.

```
HRESULT CBSPTree::SetPVSData( UCHAR PVSData[], unsigned long PVSSize, bool PVSCompressed )
{
    // Release any previous data
    if (m_pPVSData) delete[] m_pPVSData;

    try
    {
        // Allocate the PVS Set
        m_pPVSData = new UCHAR[ PVSSize ];
        if (!m_pPVSData) throw std::bad_alloc(); // VC++ Compat

        // Copy over the data
        memcpy( m_pPVSData, PVSData, PVSSize );

    } // End Try Block

    catch ( std::bad_alloc )
    {
        return BCERR_OUTOFMEMORY;

    } // End Catch Block

    // Store Values
    m_lPVSDataSize      = PVSSize;
```

```
    m_bPVSCompressed    = PVSCompressed;

    // Success
    return BC_OK;
}
```

After the new array has been allocated the PVS data is copied over from the passed array into the tree's m_pPVSData array. We also copy over the size and compression status of the data into the m_lPVSDataSize and m_bPVSCompressed member variables.

After this function has been called by the CProcessPVS module, the BSP tree will contain all relevant PVS information. This new BSP tree information will also be written out to file when the scene is saved.

That covers all the changes to the CBSPTree class so we will now look at some minor modifications that have been made to the CBSPLeaf class.

The CBSPLeaf Class - Updated

With the introduction of the portal generation and PVS calculation process in this lab project, two new member variables will be added to our leaf structure. Each leaf will now need to store a ULONG array of indices into the tree's portal array describing the portals in that array that reside in that leaf. You will recall from the accompanying text book that in addition to each portal storing the indices of the leaves in which it belongs, each leaf will store the indices of the portals (which index in to the BSP tree's portal array/vector) that reside in that leaf. We also discussed in the text book how because of the fact that the PVS data for every leaf will be combined into a single PVS data block when stored in the BSP tree (as we have seen), each leaf will need to store the index of the BYTE in the tree's m_pPVSData array where its PVS data set begins. Here is the updated class declaration for CBSPLeaf contained in the file, CBSPTree.h

Excerpt from CBSPTree.h

```
class CBSPLeaf
{
public:

    // Constructors & Destructors
             CBSPLeaf( );
    virtual ~CBSPLeaf( );

    // Public Functions for This Class
    bool            BuildFaceIndices( CBSPFace * pFaceList );
    bool            AddPortal( unsigned long PortalIndex );

    // Public Variables for This Class
    std::vector<long>   FaceIndices;    // Indices to faces in this leaf

    std::vector<long>   PortalIndices;  // Indices to portals in this leaf
    unsigned long       PVSIndex;       // Index into the master PVS array

    CBounds3            Bounds;         // Leaf Bounding Box
};
```

As you can see, there is also a new method called ‘AddPortal’ which is a simple helper function that allows the caller (in this application the portal generation module) to add a portal index to the leaf’s PortalIndices array. Let us have a look at those two new member variables first.

std::vector<long> PortalIndices;
This is an array/vector of portal indices describing the portals that reside in the leaf. Each element in this vector is an index into the CBSPTree::m_vpPortals vector. This array will be filled during the portal generation process.

unsigned long PVSIndex;
This single unsigned long member describes a byte offset into the CBSPTree::m_pPVSData where the leaf’s visibility bits begin. This is calculated and stored in each leaf during the PVS calculation process.

Constructor - CBSPLeaf

The leaf constructor now simply sets the PVSIndex of the leaf to -1 initially indicating that either no PVS data exists for the tree or that it is has not yet been calculated.

```
CBSPLeaf::CBSPLeaf()
{
    // Initialise anything we need
    PVSIndex        = -1;
}
```

AddPortal - CBSPLeaf

The CBSPLeaf::AddPortal method is called to add a portal index to the leaf’s PortalIndices array. This method is called during the portal generation process when a portal is found to exist in a leaf.

The function uses the same BSP_ARRAY_THRESHOLD strategy to minimize the number of array capacity resizes that must be performed during the portal generation process. As you can see in the following code, if the vector size reaches the vector capacity then the capacity of the vector is resized to make room for N more indices (were N is the current resize threshold). The passed portal index is then added to the end of the array.

```
bool CBSPLeaf::AddPortal( unsigned long PortalIndex )
{
    try
    {
        // Resize the vector if we need to
        if (PortalIndices.size() >= (PortalIndices.capacity() - 1))
        {
            PortalIndices.reserve( PortalIndices.size() + BSP_ARRAY_THRESHOLD );
        } // End If

        // Finally add this portal index to the list
```

```
        PortalIndices.push_back( PortalIndex );

    } // End Try Block

    catch (...)
    {
        // Clean up and bail
        PortalIndices.clear();
        return false;
    } // End Catch

    // Success
    return true;
}
```

We have finally covered all the changes to the BSP tree and the CCompiler class and are now ready to start looking at the source code to the CProcessPRT class. This is the module whose ‘Process’ function is used to generate the portals for the BSP tree.

The Portal Generator Module

The portal generation module is the first of the two new modules we will introduce in this lab project. It is vitally important that you have read the accompanying text book and especially the section that pertains to portal generation before continuing with this section. The portal generation code is highly recursive and hard to follow if you have not achieved a grasp of the theory. The theory will not be explained in this work book and it is assumed that you have at least a basic understanding of the portal generation algorithm we are using when viewing this section.

As we have discussed, the portal generation module is contained in a class called CProcessPRT. That is, this module contains all the functions that CCompiler will call to generate the portals for the BSP tree. We saw earlier, that the portal generation process is invoked from a function in CCompiler called PerformPRT which is called from CCompiler::CompileScene should portal generation be enabled for the current compile. The PerformPRT method calls a handful member functions of the CProcessPRT object that are common to all our modules. These include such trivial tasks as informing the module of the BSP tree that is being used, the logger object that error/status reports should be sent to and informing the module of the parent CCompiler object that is invoking the process. This information is all stored inside the CProcessPRT module prior to the CProcessPRT::Process method being called. This same strategy is common across all the modules. That is, for each module, we set up some of its member variables prior to calling its Process method. It is the Process method of each module that kick starts the actual process. In the case of the CProcessPRT module, it is the Process method that will generate all the portals for the tree. On function return, all portals will have been compiled and stored in the BSP tree’s portal array. All leaves in the tree will contain an array of portal indices that describe the portals that were found to reside in those leaves. Finally, each portal will contain a 2 element array of leaf indices describing the two leaves that each portal forms the doorway between. We will now examine the code to this module.

The CProcessPRT Class

The CProcessPRT class declaration is contained in the file ProcessPRT.h and is also shown below. The public interface of this class should be instantly familiar from other modules. It comprises of four methods that are exposed by other modules. The SetOptions, SetLogger and SetParent methods are common to all modules and allow the module to be configured prior to the Process method being invoked. The options structure, logging object pointer and CCompiler parent pointer passed into these methods are all stored in private members variables (you can see that they are inline functions). The private member variables are also the same as the other modules with the exception that a PRTOPTIONS structure is used to contain the portal compilation options.

Excerpt from ProcessPRT.h

```
class CProcessPRT
{
public:
    // Constructors & Destructors for This Class.
             CProcessPRT();
    virtual ~CProcessPRT();

    // Public Functions for This Class.
    HRESULT         Process( CBSPTree * pTree );
    void            SetOptions( const PRTOPTIONS& Options ) { m_OptionSet = Options; }
    void            SetLogger ( ILogger * pLogger )         { m_pLogger = pLogger; }
    void            SetParent ( CCompiler * pParent )       { m_pParent = pParent; }

private:
    // Private Functions for This Class.
    CBSPPortal     *ClipPortal          ( unsigned long Node, CBSPPortal * pPortal );
    bool            FindLeaf            ( unsigned long Leaf, unsigned long Node );
    unsigned long   ClassifyLeaf        ( unsigned long Leaf, unsigned long Node );
    HRESULT         AddPortals          ( CBSPPortal * PortalList );

    // Private Variables for This Class.
    PRTOPTIONS      m_OptionSet;        // The option set for portal Compilation.
    ILogger        *m_pLogger;          // Logging interface used to log progress etc.
    CCompiler      *m_pParent;          // Parent Compiler Pointer
    CBSPTree       *m_pTree;            // The tree used to compile the portal set.

};
```

The Process method is also no stranger to us and is the method that all modules expose to actually carry out their task. This class also has four private member functions which will be called by the public Process method to carry out its task of generating the portals and storing them in the tree.

Process - CProcessPRT

This method invoked by CCompiler to perform the portal generation. With the help of four of the class's private methods, this function is responsible for the entire process tasked to this module. That is, when this function has completed, the two way portals will all have been generated and stored in the BSP tree's portal array. Each portal in this array will also contain the indices of the two leaves in which they reside and each leaf in the BSP tree will contain an array of portal that live in that leaf.

Note: Recall from text book that while a single portal can never exist in more than two leaves, a single leaf may have many portals that reside within it.

The function takes a single parameter when called from the CCompiler object. It is passed a pointer to the BSP tree which is to have its portals generated. The passed BSP tree pointer is copied into the module's member variable (m_pTree) so that we have access to the BSP tree throughout all its functions. We then output information to the logging object via its LogWrite function displaying the message that portal compilation is about to commence. We also set a rewind marker and progress range within the logger so that every time the progress indicator is updated, we can return to the cursor position set by the rewind marker and overwrite the old progress value with the current one in the loggers output window. The initial progress value is set to zero and the range of the progress indicator is set to the number of nodes in the tree. Therefore, each time we process a node and (potentially) generate a portal for it, we can increase the current progress and have the logging object return to the rewind marker (the cursor position in the output channel where the first digit of the progress percentage will be displayed) and will update the current progress percentage value.

```
HRESULT CProcessPRT::Process( CBSPTree * pTree )
{
    HRESULT      ErrCode;
    CBounds3     PortalBounds;
    CBSPPortal * InitialPortal = NULL;
    CBSPNode   * CurrentNode   = NULL;
    CBSPNode   * RootNode      = NULL;
    CPlane3    * NodePlane     = NULL;
    CBSPPortal * PortalList    = NULL;

    // Validate values
    if (!pTree) return BCERR_INVALIDPARAMS;

    // Store tree for compilation
    m_pTree = pTree;

    try
    {
        // ************************
        // * Write Log Information *
        // ************************
        if ( m_pLogger )
        {
            m_pLogger->LogWrite( LOG_PRT,
                                 0,
                                 true,
                                 _T("Compiling scene portal information \t\t- " ) );

            m_pLogger->SetRewindMarker( LOG_PRT );
            m_pLogger->LogWrite( LOG_PRT, 0, false, _T("0%%" ) );
            m_pLogger->SetProgressRange( pTree->GetNodeCount() );
            m_pLogger->SetProgressValue( 0 );
        }
        // ************************
        // *    End of Logging     *
        // ************************
```

In the next section of the code we first test that we can retrieve the root node (node index 0) from the node array and if not then we return error (safety incase we are trying to compile portal data for a BSP

tree object that has not yet compiled its data). We fetch the root node because its bounding box will be used to create the initial portal on each node plane. We then loop through each node in the node array.

Let us now examine the contents of this node loop.

```
        // Store required values ready for use.
        if (!(RootNode = m_pTree->GetNode(0))) throw BCERR_BSP_INVALIDTREEDATA;

        // Create a portal for each node
        for (unsigned long i = 0; i < pTree->GetNodeCount(); i++)
        {
            // Update progress
            if ( m_pParent && !m_pParent->TestCompilerState()) return BC_CANCELLED;
            if ( m_pLogger ) m_pLogger->UpdateProgress( );
```

The first thing we do inside this loop is test that the parent compiler has not been put into a cancelled state by the user. If so, we simple return BC_CANCELLED because the user has obviously aborted the process mid compile. Provided this is not the case however, you can see that we instruct the logging object to update its progress as we are about to process another node.

In the next section of code we fetch the current node structure we are processing from the BSP tree's node array. We store a pointer to this node structure in the local node pointer **CurrentNode**. We then fetch from this structure the index of the node plane stored at that node so that we can fetch the node's plane from the BSP tree's plane array and store its pointer in the local variable **NodePlane.**

```
            // Store required values ready for use.
            if (!(CurrentNode = m_pTree->GetNode(i))) throw BCERR_BSP_INVALIDTREEDATA;

            if (!(NodePlane   = m_pTree->GetPlane(CurrentNode->Plane)))
                                                    throw BCERR_BSP_INVALIDTREEDATA;
```

Now that we have the node structure and the plane structure of the node we are currently processing, let us first test whether a portal could possibly exist on this node. As we know, a portal can only exist on a node plane if that node has **not** got solid space behind it. If it has then this node does not have leaves in both its half spaces and therefore, no portal generated on this node could ever bridge two leaves. When this is the case we simply skip processing this node any further and continue on to the next iteration of the loop and the next node waiting to be processed.

```
            // Skip any that have solid space behind them
            if ( CurrentNode->Back == BSP_SOLID_LEAF ) continue;
```

If we get this far without skipping to the next iteration then it means the current node we are processing has leaves on both sides and it stands a very good chance of generating a real portals. Of course, we do not know this for sure yet, but we certainly know that we will have to create a new portal on the node plane that is as large as the root node's bounding box and will have to send this portal down the tree clipping away any fragments that exist in solid space. If anything survives, then we do have a portal or multiple portal fragments that can be added to the BSP tree's portal array.

The first step is to allocate a new CBSPPortal structure using the CBSPTree::AllocBSPPortal function which we described earlier. This portal will be initially empty but we want it to represent a quad that is

located in the current node's plane and is large enough to fill the root node's bounding box. Fortunately, we have already written the CPolygon::GenerateFromPlane method that will construct such a portal given the node plane and the root nodes bounding box. Therefore, in the next section of code, you can see that after we allocate a new portal structure and retrieve the root node's bounding box, we then pass this information into the GenerateFromPlane method to generate the initial portal on that plane. We also store in the portal the index of the node on which it was initially generated in its OwnerNode member.

```
            // Allocate a new initial portal for clipping
            if (!(InitialPortal = CBSPTree::AllocBSPPortal())) throw BCERR_OUTOFMEMORY;

            // Initial Portal should fill root node
            PortalBounds = RootNode->Bounds;

            // Generate the portal polygon for the current node
            InitialPortal->GenerateFromPlane( *NodePlane, PortalBounds );
            InitialPortal->OwnerNode = i;
```

At this point we are ready to drop that initial portal in at the root node of the tree and clip it to the solid space of the tree as it makes its way down to the leaf nodes. The ClipPortal method is used for this. It is a recursive function that will call itself repeatedly until either the initial portal has been completely clipped away (in which case PortalList will be assign NULL on function return) or until it has correctly calculated the valid portal fragments in which case, they will be returned in a linked list. PortalList will point to the head of this valid portal list on function return. As the first parameter to the ClipPortal method we pass in the index of the node we would like to start clipping from which will be the root node (node zero). As the second parameter we pass our initial portal that is to be passed through the tree and clipped.

```
            // Clip the portal and obtain a list of all fragments
            PortalList = ClipPortal(0, InitialPortal );

            // Clear the initial portal value, we no longer own this
            InitialPortal = NULL;
```

Notice in the above code how when the ClipPortal method returns, we simply set the InitialPortal pointer to NULL instead of releasing it. That is because this portal will have been clipped and deleted by the ClipPortal method as it is passed through the tree and split into child fragments. As soon as we pass the initial portal into the ClipPortal method it is the responsibility of the ClipPortal method to clean up its memory when it gets split.

At this point, if PortalList is not NULL then it contains a list of one or more valid portals that have been generated on the current node plane. If this is the case these portals should be added to the BSP tree's portal array. The CProcessPRT::AddPortals method is invoked to perform this task. Contrary to the way we normally do things, we will look at the simple AddPortals method prior to examining the ClipPortal code. This will show us how the portal information returned from ClipPortal gets added to the tree and the leaves of that tree first.

```
            // Add any valid fragments to the final portal list
            if (PortalList)
            {
                if (FAILED(ErrCode = AddPortals( PortalList ))) throw ErrCode;
            } // End If PortalList
```

```
        } // Next Node

    } // End Try

    catch ( HRESULT& Error )
    {
        // If we dropped here, something failed
        if ( InitialPortal )     delete InitialPortal;
        if ( m_pLogger )         m_pLogger->ProgressFailure( LOG_PRT );
        return Error;

    } // End Catch

    // Success
    if ( m_pLogger ) m_pLogger->ProgressSuccess( LOG_PRT );
    return BC_OK;

}
```

We can see that after the AddPortals method has been called we see the closing brace to the node loop such that at the bottom of the function, every portal in the level will have been created and added to the tree's portal list. Before returning success, we output to the logging object that the mission has been a success. When the function returns, the portal generation process is over and all portals have been created and stored in the tree.

AddPortals - CProcessPRT

The AddPortals method is called from the Process method to add a list of valid portals returned from the ClipPortal method for a given node to the BSP tree's node array. The function is passed a CBSPPortal pointer that points to the head of this list. First we set up a loop to iterate through every portal in the list. The Iterator local pointer is used to step through the elements in the list.

```
HRESULT CProcessPRT::AddPortals( CBSPPortal * PortalList )
{
    unsigned long PortalIndex = 0;
    CBSPPortal  * Iterator;
    CBSPLeaf * Leaf = NULL;

    // Validate
    if (!PortalList) return BCERR_INVALIDPARAMS;

    // Iterate through the list, obtaining valid portals
    Iterator = PortalList;
    while ( Iterator != NULL )
    {
```

Inside this loop we first fetch the current portal count from the BSP tree as this will tell us the location within the BSP tree's master portal array where the next portal should be placed (at the end of the currently stored portals).

```
        // Store new portal index
        PortalIndex = m_pTree->GetPortalCount();
```

As every portal will at this point store in its LeafOwner array the indices of the two leaves in which it was found to reside during the ClipPortal function, we next loop through each of these elements and store the index of this portal (in the BSP tree's portal array) in the PortalIndices array of each leaf.

```
        // Add this portal to each leaf
        for ( int i = 0; i < 2; i++ )
        {
            Leaf = m_pTree->GetLeaf( Iterator->LeafOwner[i] );
            if (!Leaf) return BCERR_BSP_INVALIDTREEDATA;
            Leaf->AddPortal( PortalIndex );

        } // Next Leaf
```

As you can see in the above code, in each iteration of the loop we fetch the leaf index from the portal's LeafOwner array and then use that to retrieve the relevant leaf structure from the tree. Once we have a pointer to the leaf in which this portal should have its index stored, we then call the CBSPLeaf::AddPortal method (which we looked at earlier) which will add the passed index to the leaf's PortalIndices array. Remember that although we have not yet added the current portal being processed in the passed list to the BSP tree's portal array, PortalIndex describes the location of where it will be stored as this describes the current number of portals in the list prior to this portal being added.

With the portal index now stored in the two leaves in which it was found to reside, we next instruct the BSP tree to make sure there is enough space in its portal array to add this new portal pointer.

```
        // We are adding a new portal
        m_pTree->IncreasePortalCount();
```

We then finish processing the current portal by using the CBSPTree::SetPortal method to store the current portal's pointer at that index in the tree's master portal array.

```
        // Set the portal
        m_pTree->SetPortal( PortalIndex, Iterator );
```

We then assign the current portal's NextPortal member to Iterator so that in the next iteration of the loop, if NextPortal is not NULL, Iterator will point to the next portal in the passed list to be added to the tree's portal array.

```
        // Move onto next portal
        Iterator = Iterator->NextPortal;

    } // End While

    return BC_OK;
}
```

When this function returns, every portal in the passed list will have been added to the BSP tree's portal array and the leaves in which these portals reside will have had the portal indices added to their PortalIndices array.

ClipPortal - CProcessPRT

As discussed in the accompanying text book, the ClipPortal method really is the portal generation engine. It is the function that is called from the Process method and passed an initial portal that is to be clipped to the tree. When the function is first called it will visit the root node and will then recursively call itself until the portal has either been complete deleted, or until it has a list of valid portal fragments to return. Valid portals are fragments of the initial portal passed in the root that ended up in empty space and were found to reside in two leaves. This is a rather huge function which was explained in detail in the text book. As this version of the function is almost identical we will make our way through it quite quickly so that you can see the version of the function that is used by our compiler.

The function is passed a node index (which will be the root node the first time it is called from the Process method) and a portal. The first time this function is called this portal will be the initial portal that was created on the node inside the Process method, for nodes further down the tree, this portal may be a fragment of that initial portal.

```
CBSPPortal * CProcessPRT::ClipPortal( unsigned long Node, CBSPPortal * pPortal )
{
    // 52 Bytes including Parameter list (based on __thiscall declaration)
    CBSPPortal   * PortalList     = NULL, *FrontPortalList = NULL;
    CBSPPortal   * BackPortalList = NULL, *Iterator        = NULL;
    CBSPPortal   * FrontSplit     = NULL, *BackSplit       = NULL;
    CBSPNode     * CurrentNode    = NULL;
    CPlane3      * CurrentPlane   = NULL;
    unsigned long  OwnerPos, LeafIndex;

    // Validate Requirements
    if (!pPortal || !m_pTree) throw BCERR_INVALIDPARAMS;

    // Store node for quick access
    if (!(CurrentNode  = m_pTree->GetNode( Node ))) throw BCERR_BSP_INVALIDTREEDATA;
    if (!(CurrentPlane = m_pTree->GetPlane(CurrentNode->Plane)))
                                                    throw BCERR_BSP_INVALIDTREEDATA;
```

As we can see in the above section of code, we first use the passed node index to fetch the node structure from the BSP tree. This is the node that we are currently visiting with the passed portal/fragment. We then use the node's Plane index to fetch the node's plane structure from the BSP tree's master plane array.

The rest of the function is essentially just a switch statement which deals with the result of classifying the portal against the plane. The following code classifies the vertices of the polygon against the node plane. We use the CPlane3::ClassifyPoly function for this task. The function will return either CLASSIFY_FRONT, CLASSIFY_BACK, CLASSIFY_ONPLANE or CLASSIFY_SPANNING.

```
    // Classify the portal against this nodes plane
     switch (CurrentPlane->ClassifyPoly(      pPortal->Vertices,
                                              pPortal->VertexCount,
                                              sizeof(CVertex))    )
    {
```

The front and back cases are small and simple to deal with but the spanning and on plane cases are considerably more complex. We will look at the CLASSIFY_ONPLANE case first.

```
        case CLASSIFY_ONPLANE:

            // The Portal has to be sent down Both sides of the tree and tracked.
            // Send it down front first but DO NOT delete any bits that end up in solid
            // space, just ignore them.
            if (CurrentNode->Front < 0 )
            {
                        // The Front is a Leaf, determine which side of the node it fell
                LeafIndex   = abs(CurrentNode->Front + 1);
                OwnerPos    = ClassifyLeaf( LeafIndex, pPortal->OwnerNode );

                // Found the leaf below?
                if ( OwnerPos != NO_OWNER)
                {
                    // This portal is added straight to the front list
                    pPortal->LeafOwner[OwnerPos] = LeafIndex;
                    pPortal->NextPortal          = NULL;
                    FrontPortalList              = pPortal;
                    pPortal->LeafCount++;

                } // End if leaf found
                else
                {
                    delete pPortal;
                    return NULL;

                } // End If no leaf found

            } // End if child leaf
            else
            {
                    // Send the Portal Down the Front List and get returned
                    // a list of PortalFragments that survived the Front Tree
                    FrontPortalList = ClipPortal(CurrentNode->Front, pPortal);

             } // End If child node
```

The above code shows the first section of dealing with the on plane case. When a portal is on plane we need to send that portal down the front tree so that it can be clipped to the front tree of the node. This will return a linked list of any portal fragments that survive the front tree. Each portal in this list should then be passed down and clipped to the back tree of the node. Any fragment of the portal passed into this function that survives both the front and back trees of the node can then be compiled into a linked list and returned from the function.

In the above section of code we show the portion of the on plane case that deals with sending the portal down the front of the node first. If the node's Front member contains a negative number then we know that there is a leaf to the front of this node (empty space) and as such, the portal should record the index of this leaf in the portal. This is one of the leaves the portal has been found to reside in. Therefore, when this is the case we add 1 to the node's Front value and ABS it so that we have an index into the BSP tree's leaf array of the leaf in which the portal has landed. We then call the ClassifyLeaf function which will return either NO_OWNER, FRONT_OWNER or BACK_OWNER indicating whether this leaf is located in the front or back half space of the node. If the function returns NO_OWNER then it means that the leaf the portal has landed in does not exist beneath the portal's owner node in the tree and therefore this is a rogue portal (rogue portals are discussed in the text book). When this is the case we simply delete the portal as this portal is not valid for the current node being generated. However, if

FRONT_OWNER or BACK_OWNER is returned then we store the leaf index in the portal's LeafOwner array. Notice that FRONT_OWNER and BACK_OWNER are also used as the array indexes so that the leaf that exists in the front space of a portal will always be contained in the first element of the portal's LeafOwner array and the second element will always contain the index of the leaf contained in the back half space of the node. You can also see in the above code that if the portal does land in a leaf, we increase the portal's LeafCount member to reflect the fact that we have just added a leaf index to the portal's leaf array. We also assign the FrontPortalList local variable to point at this portal which will be used in a moment. We also make sure that the portal's next pointer is set to NULL.

Finally notice at the bottom of the above code, how if a leaf does not exist down the front of this node, but a child node exists instead, the ClipPortal function is called recursively to send the portal down the front tree of the current node. This function will either return a linked list of all the fragments of this portal that survived the front tree, the head of which is assigned to the FrontPortalList local variable, or will return NULL if none of the portal survived being clipped to the front tree.

At this point, FrontPortalList either points to the single portal that made it into a leaf to the front of this node, a list of fragments that survived the portal being clipped to the front tree of the node, or NULL if either the portal made it into a leaf that was not beneath the owner node in the tree (rogue portal fragment) or if the portal was clipped to the front tree and was found to be contained completely in solid space.

In the next section of code we see that if FrontPortalList equals NULL then there is nothing more to do at this node. The portal passed into this node has been completely deleted so we return NULL.

```
            // If nothing survived return here.
            if (FrontPortalList == NULL) return NULL;
```

However, if there are portals in our front list then we know each will have to be clipped to the back tree of the node next. If the node has no back child (solid space behind it) then we simply return the front list of portal fragments.

```
            //// If the back is solid, just return the front list
            if ( CurrentNode->Back == BSP_SOLID_LEAF ) return FrontPortalList;
```

Now we will loop through each portal in the front portal list and will send each one in the list into the ClipPortal function to clip it to the front tree of the node. Each time we call the ClipPortal function to send the current front list portal being processed down the back tree, we will take the returned list of portals and add them to a larger list that is being compiled. This larger list will contain, at the end of the next section of the code, any fragments that survived both the front and back trees of the node.

```
            // Loop through each front list fragment and send it down the back branch
            pPortal = FrontPortalList;

             while ( pPortal != NULL )
             {
                CBSPPortal * NextPortal = pPortal->NextPortal;
                BackPortalList          = NULL;
```

At the head of the loop we cache the next portal in the front portal list so that when the current portal we are processing gets sent down the back tree and potentially deleted, we do not loose access to the next portal in the front portal list to be processed. The variable BackPortalList will be used to retrieve any fragments of the current front portal fragment that is sent down the back tree of the node.

> **Note**: Remember that the current portal being processed in this loop (pPortal) is a portal fragment from the front portal list that is now about to be clipped to the back tree.

First we test the Back member of the current node and if found to be negative it means an empty space leaf exists to the back of this node. Here we are adding support for empty back leaves that can occur in very special cases which will be discussed later in the series. As discussed in the text book however, a level provided for solid BSP compilation, will not ever contain populated back leaves. Still, we will add support for populated back leaves now.

As we did in the case of an immediate front leaf, if a leaf does exist to the back of this node, it means the portal has landed in this leaf. As such, we convert the node's Back member into a valid leaf index and fetch the appropriate leaf structure from the BSP tree's leaf array. We then classify this leaf against the portal's owner node to determine in which half space of the portal this leaf exists. The leaf indices are then stored in the portal depending on the outcome.

```
                // Empty leaf behind?
                if ( CurrentNode->Back < 0 )
                {
                    // The back is a Leaf, determine which side of the node it fell
                    LeafIndex   = abs(CurrentNode->Back + 1);
                    OwnerPos    = ClassifyLeaf( LeafIndex, pPortal->OwnerNode );

                    // Found the leaf below?
                    if ( OwnerPos != NO_OWNER )
                    {
                        // Attach it to the back list
                        pPortal->LeafOwner[OwnerPos] = LeafIndex;
                        pPortal->NextPortal          = BackPortalList;
                        BackPortalList               = pPortal;
                        pPortal->LeafCount++;

                    } // End if leaf found
                    else
                    {
                        // Delete the portal, but continue to the next fragment
                        delete pPortal;
                        continue;

                    } // End If no leaf found

                } // End if child leaf

                else

                {
                    // Send the Portal Down the back and get returned a list of
                    // PortalFragments that survived the Front Tree
                    BackPortalList      = ClipPortal(CurrentNode->Back, pPortal);

                } // End If child node
```

Notice that if a leaf does not exist immediately to the back of this node it means a child node must exist there instead. When this is the case we send the portal down the back of the node to clip it to the node's back tree. Any surviving fragments are returned in a linked list, the head of which is assigned to the BackPortalList pointer.

At this point, BackPortalList contains only the list of fragments for a single fragment that survived the front tree of the node and as discussed, we must collect all the BackPortalList's generated from each portal in FrontPortalList and stitch them together into a single linked list that can be returned from the function. The PortalList local variable will be used to point at the head of this combined list which will be returned from the function.

In the next section of code we can see that assuming that BackPortalList is not null, we must add them to the current list we have compiled so far. We do this by first iterating to the tail of the BackPortalList.

```
                // Anything in the back list?
                if (BackPortalList != NULL)
                {
                    // Iterate to the end to get the last item in the back list
                    Iterator = BackPortalList;

                    while ( Iterator->NextPortal != NULL) Iterator = Iterator->NextPortal;
```

At this point Iterator will point to the last element in BackPortalList. We now assign the NextPortal member of this final portal in the back portal list to point at 'PortalList' which currently points to the head of the list of portal fragments we have collected so far. What we are doing is adding the back portal list to the front of the portal list of all the fragments we have collected so far. Here is the remaining code of the on plane case.

```
                    // Attach the last fragment to the first fragment
                    // from the previous iteration.
                    Iterator->NextPortal = PortalList;

                    // Portal List now points at the current complete
                    // list of fragments collected so far
                    PortalList = BackPortalList;

                } // End if BackPortalList is not empty


                // Move on to next portal
                pPortal = NextPortal;

            } // End While Portal != NULL

            // Return the full list
            return PortalList;
```

As you can see after we have assigned the Iterator to point at PortalList (the current head of the list of all fragments we have collected so far), we the reassign PortalList to point at BackPortalList so that it now points to the complete list of fragments we have collected so far, including those contained in BackPortalList. Finally, we can see that at the bottom of main loop that iterates through each portal in FrontPortalList, we assign pPortal to point at NextPortal. Next Portal is where we stored a pointer to the

next portal in FrontPortalList that will need to be clipped to the back tree of the node in the next iteration.

Outside the loop and in the very last line of code shown above, PortalList will contain all the fragments of the portal passed into the function that survived both the front and back trees of the current node being visited. This linked list is returned from the function.

With the on plane case covered, we will next look at what happens if the portal passed into the function is found to be contained entirely in the front half space of the current node being visited.

If there is a leaf immediately attached to the front of this node then it means the portal has made it into a leaf. When this is the case we see that familiar piece of code that fetches the leaf from the BSP tree's leaf array and classifies it against the owner node of the portal. Depending on which side of the owner node's plane the leaf is found to reside, the leaf index is recorded in the appropriate position in the portals LeafOwner array and the portals leaf count is increased so that we know how many leaves this portal has been found to exist in at this point. Below we show the entire CLASSIFY_INFRONT case.

```
            case CLASSIFY_INFRONT:

             // Either send it down the front tree or add it to the portal
             // list because it has come out in Empty Space
            if (CurrentNode->Front < 0 )
            {
                // The front is a Leaf, determine which side of the node it fell
                LeafIndex   = abs(CurrentNode->Front + 1);
                OwnerPos    = ClassifyLeaf( LeafIndex, pPortal->OwnerNode );

                // Found the leaf below?
                if ( OwnerPos != NO_OWNER )
                {
                    // This is just returned straight away, it's in an empty leaf
                    pPortal->LeafOwner[OwnerPos] = LeafIndex;
                    pPortal->NextPortal          = NULL;
                    pPortal->LeafCount++;
                    return pPortal;

                } // End if leaf found
                else
                {
                    delete pPortal;
                    return NULL;

                } // End if leaf not found

            } // End if child leaf
            else
            {
                // Pass down the front
                PortalList = ClipPortal(CurrentNode->Front, pPortal);
                return PortalList;

            } // End If child node

            break;
```

Above we can see that if a leaf does not exist immediately to the front of this node it means a child node exists and as such, the portal is passed down the front tree of the node and clipped to any solid space that may exist there. The ClipPortal method will return a list of one or more fragments of this portal that survived being clipped to the front tree which are then returned from the function to the parent instance of the function.

The CLASSIFY_BEHIND case is almost the same except with a very important difference. If solid space exists to the back of the node then the portal has to be deleted and NULL returned. It is this case during the recursive process that is responsible for clipping away the parts of a portal that land in solid space.

```
            case CLASSIFY_BEHIND:

            // Test the contents of the back child
            if (CurrentNode->Back == BSP_SOLID_LEAF )
            {
                // Destroy the portal
                delete pPortal;
                return NULL;

            } // End if solid leaf
```

However, if solid space does not exist down the back of this node then two other conditions may be true. Either an empty space leaf exists down the back of this node in which case we store the leaf index in the portal in the normal way, or a child node exists down the back of the current node in which case the portal must be clipped to the back tree and the resulting fragment list returned as shown below.

```
            else

            if (CurrentNode->Back < 0 )
            {
                // The back is a Leaf, determine which side of the node it fell
                LeafIndex   = abs(CurrentNode->Back + 1);
                OwnerPos    = ClassifyLeaf( LeafIndex, pPortal->OwnerNode );

                // Found the leaf below?
                if ( OwnerPos != NO_OWNER )
                {
                    // This is just returned straight away, it's in an empty leaf
                    pPortal->LeafOwner[OwnerPos] = LeafIndex;
                    pPortal->NextPortal          = NULL;
                    pPortal->LeafCount++;
                    return pPortal;

                } // End if leaf found
                else
                {
                    delete pPortal;
                    return NULL;

                } // End if leaf not found

            } // End if child leaf
            else
            {
                // Pass down the back
```

```
                PortalList = ClipPortal(CurrentNode->Back, pPortal);
                return PortalList;

            } // End If child node

            break;
```

The final case we must deal with I this function is the case where the portal is spanning the plane. When this is the case we must split the portal into two child fragments and clip the back fragment to the back tree of the node and the front fragment to the front tree of the node. Any surviving fragments from the front split portal and the back split portal are joined together into a single linked list which is then returned from the function. We will look at this case a section at a time.

```
            case CLASSIFY_SPANNING:

            // Allocate new front fragment
            if (!(FrontSplit = CBSPTree::AllocBSPPortal())) throw BCERR_OUTOFMEMORY;

            // Allocate new back fragment
            if (!(BackSplit  = CBSPTree::AllocBSPPortal())) { delete FrontSplit;
                                                              throw BCERR_OUTOFMEMORY; }

            // Portal fragment is spanning the plane, so it must be split
            if (FAILED( pPortal->Split(*CurrentPlane, FrontSplit, BackSplit)))
            {
                delete FrontSplit; delete BackSplit;
                throw BCERR_OUTOFMEMORY;
            } // End If

            // Delete the ORIGINAL portal fragment
            delete pPortal;
            pPortal = NULL;
```

As the above code shows, because we know the portal is spanning the plane it will have to be split into two children so we first allocate two new empty CBSPPortal structures. These are then passed into the parent portal's Split method along with the plane of the current node we are visiting. When the split function returns, FrontSplit will contain the fragment of the portal that exists in the front half space of the passed plane and BackSplit will contain the fragment that exists in the back half space. The original portal (pPortal) that was passed into the function can now be deleted as it has been replaced by these two splits.

Our next task is to deal with the front split first by sending it down the front tree of the node. If a leaf exists immediately to the front of the node then the front split portal obviously exists in this leaf. When this is the case the leaf index is recorded in the portal as we have seen many times before.

```
            // There is another Front NODE ?
            if (CurrentNode->Front < 0 )
            {
                // The front is a Leaf, determine which side of the node it fell
                LeafIndex   = abs(CurrentNode->Front + 1);
                OwnerPos    = ClassifyLeaf(LeafIndex, FrontSplit->OwnerNode );

                // Found the leaf?
                if ( OwnerPos != NO_OWNER)
                {
```

```
                FrontSplit->LeafOwner[OwnerPos] = LeafIndex;
                FrontSplit->NextPortal          = NULL;
                FrontPortalList                 = FrontSplit;
                FrontSplit->LeafCount++;

            } // End if leaf found
            else
            {
                delete FrontSplit;

            } // End If no leaf found

        } // End if child leaf
        else
        {
                FrontPortalList = ClipPortal(CurrentNode->Front, FrontSplit);

        } // End If child node
```

Notice in the **else** case however that if a child node exists here instead, the front split portal is clipped to the front tree of the node with any surviving fragments being assigned to the FrontPortalList local variable. Notice that even if the portal makes it into the leaf, we also assign FrontPortalList to point at it so that regardless of whether a node or a leaf exists to the front of this node, we know that FrontPortalList will point to one or more portals that have survived the front tree of the node at this point.

With the front split being clipped to the front tree dealt with, we will now send the back split portal down the back tree of the node. If there is solid space behind this node then the back split has landed in solid space and is therefore simply deleted. Alternatively, if a leaf exists to the back of this node we see that familiar code that classifies the leaf against the node and stores the leaf index in the appropriate location in the portals LeafOwner array.

```
        // There is another back NODE ?
        if ( CurrentNode->Back == BSP_SOLID_LEAF )
        {
            // We ended up in solid space
            delete BackSplit;

        } // End if solid leaf
        else if (CurrentNode->Back < 0 )
        {
            // The back is a Leaf, determine which side of the node it fell
            LeafIndex   = abs(CurrentNode->Back + 1);
            OwnerPos    = ClassifyLeaf(LeafIndex, BackSplit->OwnerNode );

            // Found the leaf?
            if ( OwnerPos != NO_OWNER)
            {
                BackSplit->LeafOwner[OwnerPos] = LeafIndex;
                BackSplit->NextPortal          = NULL;
                BackPortalList                 = BackSplit;
                BackSplit->LeafCount++;

            } // End if leaf found
            else
            {
                delete BackSplit;
```

```
            } // End If no leaf found

        } // End if child leaf
```

Finally, if a child node exists to the back of this node instead of a leaf, we clip the back split portal to the back tree of the node using the BackPortalList local pointer to point to any surviving fragments on function return.

```
    else
        {
            BackPortalList = ClipPortal(CurrentNode->Back, BackSplit);

        } // End If child node
```

Notice that even in the case where the portal makes it into a back leaf, we assign BackPortalList to point at the portal so regardless of whether or not it landed in a leaf or was clipped to the back of the tree, BackPortalList will contain any fragments of the back split portal that survived the back node of the tree.

At this point we have FrontPortalList and BackPortalList which can potentially contain the fragments of the front split portal and the back split portal that survived being sent down the front and back of the node respectively. Our final step is two join these two portal lists together into a single list before returning this combined list from the function.

The following code shows that if there are portals in FrontPortalList then we iterate through the list to get a pointer to the last portal in that list. If BackPortalList isn't NULL then we assign the NextPortal member of that last portal in FrontPortalList to point to the first portal in the back portal list which is then returned from the function.

```
        // Find the End of the front list and attach it to Back List
        if (FrontPortalList != NULL)
        {
                // There is something in the front list
                Iterator = FrontPortalList;

                while (Iterator->NextPortal != NULL) Iterator = Iterator->NextPortal;

                if (BackPortalList != NULL) Iterator->NextPortal = BackPortalList;
                    return FrontPortalList;

        } // End if front list
```

If there are no portals in the front portal list then alternatively we just return either the back portal list or NULL if no portals exist in this the back portal list either as shown below.

```
        else
        {
            // There is nothing in the front list simply return the back list
            if (BackPortalList != NULL) return BackPortalList;
            return NULL;

        } // End if no front list
```

```
            // If we got here, we are fresh out of portal fragments so simply return NULL.
            return NULL;

    } // End switch

    return NULL;
}
```

The ClipPortal method is certainly an intimidating function on first appearance although, examining the various cases in isolation has shown that this is really just a special case CSG function. Most of the code that seems to make the function look overly complex is actually trivial linked list manipulation and management code. This function is the core of the portal generation process. As we have seen, it is called by the CProcessPRT::ClipPortal method for each initial portal that is created on a node plane.

Whenever a portal is found to exist in a leaf in the above code, we record the index of that leaf in the portal. Whether the leaf index is stored in the first element of the portals LeafOwner array or the second depends on whether the leaf exists in the front or back half space of the node respectively. To make this determination the ClassifyLeaf method is used. This method also takes care of identifying rogue portal fragments if the leaf can not be found down either the front or back tree of the current portals owner node. Let us have a look at the code to this function next.

ClassifyLeaf - CProcessPRT

When this function is called from ClipPortal it is passed the index of the leaf in which the portal fragment has been found to reside and is also passed the index of the portal's owner node. Recall that the owner node is the node for which the initial portal was created and sent into the first instance of the ClipPortal method.

The function first uses the passed node index to fetch the node structure from the BSP tree's node array.

```
unsigned long CProcessPRT::ClassifyLeaf( unsigned long Leaf, unsigned long Node )
{
    CBSPNode * CurrentNode = NULL;

    // Validate Requirements
    if (!m_pTree) throw BCERR_INVALIDPARAMS;
    if (!(CurrentNode = m_pTree->GetNode( Node ))) throw BCERR_BSP_INVALIDTREEDATA;
```

If the node has a negative number in its Front member then it means a leaf exists down the front of the node. In this case we convert it to a valid leaf index and perform an equality test with the passed leaf index. If they are equal then we have located the passed leaf as being attached immediately to the front of the node. This means the leaf we are looking for is in the front half space of the portal so we return FRONT_OWNER.

```
    // Check to see if the front is this leaf
    if ( CurrentNode->Front < 0 )
    {
        if ( abs(CurrentNode->Front + 1) == Leaf ) return FRONT_OWNER;
    }
```

If the node's Front member is non negative then it contains the index of the front child node. When this is the case we use the CProcessPRT::FindLeaf function (discussed in a moment) to traverse the front tree of the node looking for the leaf. If the function returns true then the leaf was located down the front tree of the node so we can also return FRONT_OWNER. If the function did not return true then we were unable to locate the leaf in the front tree so we will have to search the back tree of the node instead.

```
    else
    {
        if (FindLeaf( Leaf, CurrentNode->Front )) return FRONT_OWNER;
    } // End If
```

Provided that the node's Back member is non negative it means a child node exists there so we should traverse down the back tree of the node searching for the leaf using the FindLeaf function once again. If the function returns true the leaf was located in the back tree which means this is the leaf that exists in the back space of the portal thus we return BACK_OWNER. If none of these cases are true then it means the portal fragment obviously landed in a leaf that is not beneath the portal's owner node in the tree and as such is obviously a rogue fragment that should be deleted. When this is the case we reach the bottom of the function where NO_OWNER is returned.

```
    if ( CurrentNode->Back >= 0 )
    {
        if (FindLeaf(Leaf, CurrentNode->Back )) return BACK_OWNER;
    } // End If

    return NO_OWNER;
}
```

When NO_OWNER is returned from this function back to the ClipPortal method, the portal fragment is deleted.

FindLeaf - CProcessPRT

This is the recursive leaf searching function that was called from the previously discussed function to search for a given leaf down a given sub-tree. As we saw in the previous function, the first parameter to this function is where the index of the leaf we wish to search for should be passed. The second parameter is where the index of the node we would like to start searching from should be passed. As we saw in the previous function, this is the back or front child of the portal's owner node for which the leaf search is being performed.

The function first uses the passed node index to fetch the relevant node structure from the tree's node array.

```
bool CProcessPRT::FindLeaf( unsigned long Leaf, unsigned long Node )
{
    CBSPNode * CurrentNode = NULL;

    // Validate Requirements
    if (!m_pTree) throw BCERR_INVALIDPARAMS;
    if (!(CurrentNode = m_pTree->GetNode( Node ))) throw BCERR_BSP_INVALIDTREEDATA;
```

If a leaf exists down the front of this node then we test to see if that leaf has the same index as the leaf we are searching for. If so, we have found our leaf so we return true.

```
    // Check to see if the front is this leaf
    if ( CurrentNode->Front < 0 )
    {
        if ( abs(CurrentNode->Front + 1) == Leaf ) return true;
    }
```

If a leaf does not exist down the front of the current node we are visiting but a child node exists there instead, we will recur down the front tree of this node searching for the leaf.

```
    else
    {
        if (FindLeaf( Leaf, CurrentNode->Front )) return true;
    } // End If
```

If we get this far then it means the leaf could not be found down the front tree of the current node so we will search the back tree of the current node instead and return true if it is located.

```
    // Iterate down the back if it's a node
    if ( CurrentNode->Back >= 0 )
    {
        if (FindLeaf( Leaf, CurrentNode->Back )) return true;
    } // End If

    return false;
}
```

If we reach the bottom of the function without returning true then it means the leaf could not be found down the front and back tree of the current node so we return false.

Portal Generation Conclusion

This completes our coverage of the portal generation module and as we have seen, it is implemented via a handful of functions. We have also seen that all our CCompiler object had to do was invoke the CProcessPRT::Process method to populate the BSP with portal data.

We are at the mid-way point in this lab project having implemented one of the two modules necessary to add potential visibility set calculation to our compiler application. In the next section we will discuss the implementation of the CProcessPVS class which uses the portal data now present in the tree to perform the final calculation of the PVS data. It should be noted that the portal generation module must be enabled for the PVS calculation module to work. As the PVS calculation module can not possibly perform its task with our portal data being present in the tree, the module will terminate immediately if no portal data is found to be present in the BSP tree.

The PVS Calculator

It is crucial that you have read the PVS section in the accompanying text book before continuing with this section of the work book. In this section we will not re-cover the theory of PVS calculation and anti-penumbra generation and clipping. It is assumed you have read and understood the text book and are now interested in seeing how this technique pertains to this particular application and our application structures and classes.

Our PVS calculator is contained in the module CProcessPVS. It is this module's Process method which is called from CCompiler post portal generation. The PVS calculation module itself has all its data and structures contained in the project source files ProcessPVS.h and ProcessPVS.cpp.

In ProcessPVS.h we have a compiler define named PVS_COMPRESSDATA which can be set to zero or one to control whether the resulting PVS data should be zero run length encoded by the PVS generation module prior to being stored in the BSP tree.

```
#define PVS_COMPRESSDATA        1       // 1 = ZRLE Compress, 0 = Don't Compress
```

By default we set this to 1 and it is unlikely that you will want to change this. PVS data sets for complex levels can be quite large if no compression is used and while memory is fairly abundant on today's end user systems, we do not want to waste it considering the typical huge number of other resources that may have to be stored. Also, we discussed in the text book how zero run length encoding our data actually helps speed up the rendering of a given leaf's PVS by allowing us to skip past entire runs of non-visible leaves with a single byte increment of the PVS data pointer.

Before we examine the code to the CProcessPVS module, there are several support structures used by this process that we must first discuss. For example, we know that the two-way portals stored in the BSP tree will have to be duplicated into a number of one way portal structures that have added member variables that pertain to the PVS calculation process. We also discussed in the text book how in order to reduce memory allocation and fragmentation during the recursive clipping process, each of these portals will have the ability to share its underlying geometry with other portals.

The CPVSPortal Class

The CPVSPortal class is the object that will be used to store the one way portals used by the PVS calculation module. For each original two-way portal stored in the BSP tree at the end of the portal generation process, two CPVSPortal objects will be created. Each of the two one-way portals will share and represent the same geometry of the two-way portal from which they were cloned but each will point into an opposing half space. In the text book we had a section devoted to the need for us having a strict portal flow during the PVS clipping process and that is what these one-way portals provide us.

Each one-way portal which is temporarily generated for the PVS calculation module stores much more additional information than a normal two-way portal it was cloned from. We have to store in that portal the index of its neighbor leaf. The neighbor leaf is the leaf in which the one-way portal does not reside but has its normal facing into. That is to say, a one-way portal's visibility flows out from its owner leaf

into the neighbor leaf. We will also need to store in the one-way portal the visibility bit sets that are being compiled for that portal. Recall from the text book, that the PVS for the tree is actually generated by first calculating the PVS of each portal. Therefore, each of these portals will need to have a member where we can store its PVS. Each leaf's PVS is then calculated by simply accumulating the visibility information of each portal that resides within that leaf.

The CPVSPortal structure is shown below followed by a description of its members.

Excerpt from CPVSPortal.h

```
class CPVSPortal
{
public:
    // Constructors / Destructors for this Class
              CPVSPortal( );
    virtual ~CPVSPortal( );

    // Public Variables for This Class
    UCHAR             Status;               // The compilation status of this portal
    UCHAR             Side;                 // Which direction does this portal point
    long              Plane;                // The plane on which this portal lies
    long              NeighbourLeaf;        // The leaf into which this portal points
    long              PossibleVisCount;     // The size of the PossibleVis array
    UCHAR            *PossibleVis;          // "Possible" visibility information
    UCHAR            *ActualVis;            // "Actual" visibility information
    bool              OwnsPoints;           // Does this own the points ??
    CPortalPoints    *Points;               // The vertices making up this portal

};
```

UCHAR Status;

This member is used to store the current status of the portal. This allows us to determine whether this portal has had its PVS calculated yet, whether it is still waiting to have its PVS calculated or whether it is currently in the process of having its visibility information calculated. During the calculation of the PVS, each portal can be set to one of the following #defines from the file CProcessPVS.h

Excerpt from CProcessPVS.h

```
#define PS_NOTPROCESSED     0       // Portal has not yet been processed
#define PS_PROCESSING       1       // Portal is currently being processed
#define PS_PROCESSED        2       // Portal has been processed
```

These status flags are used in a number of places. Firstly, as the PVS calculation process is one of essentially looping through each portal and calculating its PVS, we need to know which portals have been processed already so that we do not try and calculate its PVS more than one. Now this would not be necessary if we were to simply loop through the array of one-portals and calculate the PVS for each one in that order however, as discussed in the text book, the PVS calculator can be speeded up slightly by processing less complex portals first. That is, portals which have a lower 'Possible Visibility Count'. The possible visibility count is calculated prior to the core clipping process and describes the number of leaves that had the potential to be visible as determined by a simple flood fill. By choosing portals with the lowest possible visibility count first and which have not yet be processed (obviously), we have a situation where during the main clipping process, the calculation time of visibility information for more complex portals can be reduced by using the PVS already calculated for each of the lesser complex

portals it can see. There will only ever be one CPVSPortal with a status of PS_PROCESSING at any one time during the PVS calculation procedure.

long Plane;
In this member we will store the index of the plane in the BSP tree's plane array on which the portal was created. This is the index of the owner node of the two way portal from which this portal was cloned. As two one-way portals will be generated from a single two-way portal, both of these portals will share the same plane index. It might seem strange that these two one-way portals would share the same plane when they face into opposing half spaces of that plane but we will see that it is precisely for this reason the 'Side' member of this structure (discussed next) is used.

The plane is stored in the one-way portal because it will be needed for clipping. For example, we discussed in the text book how if during the core clipping process the generator portal is found to span the source portal's plane, the generator portal should be clipped to that plane such that any fragment of the generator portal that lay in the back space of that plane is discarded. This helps narrow the anti-penumbra and is also more vital than a simple optimization. If this was not done then when we recur through that generator portal such that it becomes the target portal in the next recursion, the anti-penumbra generated could become twisted or inverted causing erroneous clipping and endless recursive loops to be formed.

The problem one might see at the moment is that both the one-way portals duplicated from a single two-way portal share the same plane while the portals themselves are pointing into opposing half spaces of that plane. Therefore, this would only work correctly for the one-way portal that shares the same front space with the plane. For the other one-way portal, clipping the generator portal to its plane would actually remove the section of the generator portal that is in the front space of the portal instead of in its back space. This would obviously be completely incorrect which is why the 'Side' member (discussed next) is used. The Side member tells the PVS processing module on which side of the plane the portal's normal is assumed to be pointing into. If the Side member indicates that the one-way portal is actually pointing into the opposing half space of its stored plane, then the plane is flipped temporarily on the fly prior to the clip being performed such that in both cases, we always remove the section of the generator portal that is in the back space of the portal.

UCHAR Side;
As discussed above, this member is used to indicate whether the portal is pointing into the same front space as its stored node plane. If this member is set to FRONT_OWNER then it is and any generator portals can be clipped against this plane without any problems. As polygon clipping functions typically remove the polygon fragment that lay in the back space of the plane, this is correct and the fragment of the generator portal that lay in the back space of the portal will be discarded. If this member is set to BACK_OWNER then it means the stored node plane is actually facing into the opposing half space to that of the plane normal and as such, the plane direction will need to be flipped prior to the clip.

long NeighbourLeaf;
This member stores the index of the leaf that the portal flows into. As discussed in the text book, portal visibility flow happens from a source leaf, through the back of its contained portals and into their neighbor leaves. This stores the neighbor leaf of the portal; the leaf that you will arrive in if you step through the back of the portal from its owner leaf.

UCHAR *PossibleVis;

To optimize the core clipping process as much as possible, prior to the main anti-penumbra clipping processor being invoked, a very approximate PVS will be calculated using a simple flood fill technique controlled by the rules of one-way portal flow. This process will be very quick to perform and will generate a leaf visibility bit set for each portal. Any bits set to zero in this bit set represent leaves that could not possibly ever be visible from the portal and as such, we know during the core clipping process not to visit these leaves unnecessarily and perform wasteful expensive clipping operations to essentially arrive at a non visible result.

After the portal flow procedure has been perform for this portal (prior to the core clipping procedure being invoked for this portal), this unsigned char pointer will point to an array of bytes that contain the 'Possible' visibility set for the portal. Each bit in the array will represent a leaf and as such, each byte element in this array represents the visibility information for eight leaves with respect to this portal. This visibility set will be very approximate and vastly over generous but will help optimize the core recursive clipping procedure.

> **Note**: We discussed earlier how one of the compile time options for our CProcessPVS module is the PVSOPTIONS::FullCompile Boolean. If set to false, our PVS calculator will not bother performing the core clipping procedure at all and will simply return the PVS data based into the possible visibility information stored in this byte array for each portal. That is, the PVS calculated for each leaf will be the accumulation of the PossibleVis arrays of each portal contained in that leaf. While this will generate an very over generous PVS with very approximate visibility, it will compile extremely quickly which might be useful during development time when you wish to test the compiler but not wait hours for the compiler to calculate the actual potential visibility set.

long PossibleVisCount;

This member will be calculated during the initial flood of this portal and the calculation of its PossibleVis array. It will describe the number of possibly visible leaves. That is, the number of bits in the PossibleVis array that have been set to 1. As discussed a moment ago, to speed up compilation we will wish to calculate the PVS for the least complex portals first. This member describes the portals complexity and therefore, describes the order in which it should be chosen for full PVS calculation. The lower this number, the earlier in the process this portal will be chosen to have its PVS calculated.

UCHAR *ActualVis;

The job of the core clipping process of this module is to take the PossibleVis array of a portal and refine it using anti-penumbra clipping to generate the tightest potential visibility set possible. The resulting (actual) PVS for the portal will be stored in this array. That is, after this portal has been processed, the ActualVis array will contain the 'real' visibility information for this portal and the visibility information that will ultimately contribute to its owner leaf's PVS at the end of the process.

CPortalPoints *Points;

This member is of a type we have not yet discussed. The CPortalPoints structure is a specialized structure derived from CPolygon and as such is actually used to store the geometry of the portal itself. We can think of the CPortalPoints class as being a CPolygon object with a few extra member variables that pertain to the PVS clipping process.

bool OwnsPoints;
To save memory, we try to maximize the re-use of geometry among the portals. For example, we know that when we generate two one-way portals from a two-way portal that it would be a waste to allocate two CPortalPoints objects for each one-way portal. These polygons and their geometry will essentially be exactly the same as each other and will contain the exact same vertex data. Therefore, we can allocate one CPortalPoints structure (remember this is just a CPolygon derived object) and can have both one-way portals point at it. Therefore, in order to make sure that when each one-way portal is deleted we do not try and delete this shared CPortalPoints structure twice, only one of the one-way portals will have its OwnsPoints set to true. This will be assumed to be the portal that owns the structure and the one that will take care of releasing it when it is deleted. When the other one-way portal is deleted, which has this member set to false, it will not attempt to delete the CPortalPoints structure that it references as it knows that another object will handle its clean up.

Let us now have a look at the methods of this one way portal class for which there is only a constructor and a destructor.

Constructor - CPVSPortal

The constructor simply initializes all members to zero, NULL or false and sets the status of this portal to its default state of PS_NOTPROCESSED. That is, this portal has not yet had its PVS data (ActualVis array) calculated.

```
CPVSPortal::CPVSPortal()
{
    // Initialise any class specific items
    Status              = PS_NOTPROCESSED;
    Plane               = -1;
    NeighbourLeaf       = -1;
    PossibleVisCount    = 0;
    PossibleVis         = NULL;
    ActualVis           = NULL;
    Points              = NULL;
    OwnsPoints          = false;
}
```

Destructor - CPVSPortal

The destructor is simple also but sheds some light on the OwnsPoints member that we discussed a moment ago.

The portal contains three possible memory allocations that it may be responsible for releasing. If its ActualVis and PossibleVis arrays have been allocated then they will need to be deleted. Also, if this portal is the owner of the CPortalPoints object that it references then it should deleted that too.

```
CPVSPortal::~CPVSPortal()
{
    // Clean up after ourselves
    if (ActualVis && PossibleVis != ActualVis) delete []ActualVis;
    if (PossibleVis) delete []PossibleVis;
    if (Points && OwnsPoints ) delete Points;

    // Empty pointers
    PossibleVis = NULL;
    ActualVis   = NULL;
    Points      = NULL;
}
```

As you can see in the above code, we only release the CPortalPoints object if the portal's OwnsPoints Boolean is set to true.

Let us now look at the CPortalPoints object which is derived from CPolygon and contains the actual geometry of the portal referenced by this CPVSPortal object.

The CPortalPoints Class

The CPortalPoints class essentially encapsulates a portal polygon. It is derived from CPolygon and as such inherits its geometry members and methods. This class adds some of its own members and implements its own versions of the Split and Clip functions. Once again, do not worry we do not have to write polygon splitting and clipping functions all over again. These functions are simple wrappers around their base class counterparts that facilitate the copying of the extra data into the child polygons resulting from the split/clip.

This class is declared in CProcessPRT.h and is shown below. It will be followed by a discussion of its members and an examination of its member functions.

```
class CPortalPoints : public CPolygon
{
public:

    // Constructors / Destructors for this Class
             CPortalPoints( );
             CPortalPoints( const CPolygon * pPolygon, bool Duplicate = false );
    virtual ~CPortalPoints( );

    // Public Functions for This Class
    CPortalPoints *     Clip( const CPlane3& Plane, bool KeepOnPlane );
    virtual HRESULT     Split( const CPlane3& Plane,
                               CPortalPoints * FrontSplit,
                               CPortalPoints * BackSplit);

    // Public Variables for This Class
    bool                OwnsVertices;           // Do we own the vertices stored here ?
    CPVSPortal         *OwnerPortal;            // Pointer to this points parent portal ;)
};
```

As you can see, we have added only two member variables to that of those inherited from CPolygon.

bool OwnsVertices;
This member is analogous to the OwnsPoints member of the CPVSPortal structure. It allows multiple CPortalPoints structures to share the same underlying vertex data. For example, when two CPVSPortals are first created (from a given two-way portal) we have seen that one CPortalPoints structure is created which is shared by both. Furthermore, as the geometry of this portal is identical to that of the two-way portal stored in the tree, we can simply assign its vertex pointer to the vertex array stored in the two-way portal in the BSP tree. That is, for each one-way portal that we initially create, its CPortalPoints structure will not have allocated its own vertex data but will point to the vertex array of the CBSPPortal from which it was cloned.

This may all sound a little over cautious but it is quite necessary for both the compiler's performance and addressing memory footprint issues. For example, we know that if portal gets clipped then we will have to allocate a new set of vertex data for the child split fragments. There is nothing we can do about that. However, there may be many times during the process where a generator portal does not get clipped at all and as such, we can happily use the vertex data that was originally created for the CBSPPortal version of the portal.

This Boolean lets the CPortalPoints destructor know whether or not the vertex array store here is owed by (was allocated for) this portal specifically (such as if this polygon was the result of a clip operation) or if its vertex pointer is assigned to the vertex array of another object in which case the memory should not be released. As discussed, when each one-way portal is created, each of their CPortalPoints objects will not contain there own vertex data but alias the vertex data stored in the portals of the BSP tree.

CPVSPortal *OwnerPortal;
This member is used to point at the CPVSPortal that owns this object and will be responsible for its clean up. For each pair of one-way portals that we generate, only one of them will own the CPortalPoints structure that they both alias (phew, that is a lot to keep track of).

Constructor - CPortalPoints

There are two constructors for this class. The default constructor simply sets the owner portal pointer to NULL and sets the OwnsVertices Boolean to false by default as shown below.

```
CPortalPoints::CPortalPoints()
{
    // Initialise any class specific items
    OwnsVertices = false;
    OwnerPortal  = NULL;
}
```

The second constructor is a copy constructor that can be used to create and populate a new CPortalPoints object from the data stored in a passed CPolygon object.

The first parameter to the copy constructor is a pointer to the CPolygon we would like this object to copy or alias the vertex data of. The second parameter is a Boolean which specifies whether we would

like this object to allocate its own vertex array and copy the vertex data over from the passed CPolygon, or whether we would like to simply alias the vertex data by assigning the vertex data pointer to point at the vertex array of the passed CPolygon. In the later case, the CPortalPoints object will not own the vertex data and should no delete it within its destructor. We inform the destructor of whether or not the vertex data of this object should be released during object deletion by setting the OwnsVertices member to true or false respectively.

Here is the code:

```
CPortalPoints::CPortalPoints( const CPolygon * pPolygon, bool Duplicate )
{
    // Initialise any class specific items
    OwnsVertices = false;
    OwnerPortal  = NULL;
    if (!pPolygon) return;

    // Store or duplicate verts
    if ( Duplicate )
    {
        // Duplicate the vertices
        if (AddVertices( pPolygon->VertexCount ) < 0) throw BCERR_OUTOFMEMORY;
        memcpy( Vertices, pPolygon->Vertices, VertexCount * sizeof(CVertex) );
        OwnsVertices = true;

    } // End if Duplicate
    else
    {
        // Simply store a copy of the pointer info
        Vertices     = pPolygon->Vertices;
        VertexCount  = pPolygon->VertexCount;
        OwnsVertices = false;

    } // End if !Duplicate
}
```

As you can see, if the Duplicate Boolean parameter is set to true then we do indeed allocate a vertex array for this object and copy over the vertex data from the passed CPolygon. We then set the OwnsVertices member to true which will instruct the destructor to release this memory on object de-allocation. If the Boolean is set to false then we simply copy over the vertex count from the passed polygon and assign the vertex pointer to point at the passed CPolygon vertex array. We also set the OwnsVertices member to false so that we do not try to delete this vertex array on object destruction. These are not our vertices to delete.

Destructor - CPortalPoints

The destructor only deletes the vertex array of the CPortalPoints object if it owns them. That is, if the OwnsVertices Boolean is set to true.

```
CPortalPoints::~CPortalPoints()
{
    // Clean up after ourselves only if required
    if (OwnsVertices) ReleaseVertices();
```

```
    // Simply NULL our vertex values
    Vertices    = NULL;
    VertexCount = 0;
}
```

It will become more apparent why we try to share as much data as possible during the core clipping process.

Split - CPortalPoints

The CPortalPoints object implements it own polygon splitter. As with the base class version of this function it takes three parameters with the first being the split plane. The final two parameters however are now of type CPortalPoints.

As CPortalPoints is derived from CPolygon we can use the base class version of the function to perform that actual splitting of the geometry into the front and back children.

```
HRESULT CPortalPoints::Split( const CPlane3& Plane,
                                    CPortalPoints * FrontSplit,
                                    CPortalPoints * BackSplit)
{
    // Call base class implementation
    HRESULT ErrCode = CPolygon::Split( Plane, FrontSplit, BackSplit );
    if (FAILED(ErrCode)) return ErrCode;
```

As we have created two new polygons by copying over portions of vertex data from the parent polygon, each one will own its own vertices so we must set the OwnsVertices member of both the front and back split (if they exist) to true.

```
    // Copy remaining values
    if (FrontSplit)
    {
        FrontSplit->OwnsVertices = true;
    } // End If

    if (BackSplit)
    {
        BackSplit->OwnsVertices  = true;
    } // End If

    // Success
    return BC_OK;
}
```

As we have seen, this function is a simple wrapper around a call to the base class version of the function with the added logic of making sure that the child split fragments understand that they have had there own unique vertex data generated by the clipping process. This is especially true as we would not want the vertex data of the children to be deleted when the parent polygon is deleted because as we know, one of the first things we do after splitting a polygon into two children is delete the original. Our split routines (as we have seen) will always create polygon fragments that own there own vertex data.

Clip - CPortalPoints

The Clip method simply classifies the polygon represented by this object against the plane passed in as the first parameter and removes the portion of the polygon that is found to be behind the plane. The second parameter indicates whether we would like the polygon to be clipped or kept if it is found to exist on the plane itself. This will be used later during the anti-penumbra clipping process. The function returns a pointer to the new clipped polygon.

First we classify the polygon against the plane and then enter a switch statement that chooses the outcome based on the classification result. Notice at the top of the function how the NewPoints local pointer is allocated. This will be used to point to the clipped fragment and will be the pointer that is returned from the function.

```
CPortalPoints * CPortalPoints::Clip( const CPlane3& Plane, bool KeepOnPlane )
{
    CPortalPoints * NewPoints = NULL;

    try
    {
        // Classify the points
        CLASSIFYTYPE Location = Plane.ClassifyPoly( Vertices, VertexCount, sizeof(CVertex) );

        // What location ?
        switch ( Location )
        {
```

If the polygon is found to be contained in the front space of the plane then nothing is to be clipped. Therefore, we simple assign the NewPoints pointer to point at the current polygon. This means, the method will return a pointer to the CPortalPoints structure for which it was invoked. That is, the object will just return a pointer to itself.

```
            case CLASSIFY_INFRONT:
                // All were in front, simply return this
                NewPoints = this;
                break;
```

If the polygon is found to exist entirely in the back space of the clip plane then the polygon should be totally clipped away. When this is the case we return NULL indicating that none of the polygon should survive. The caller can then choose to delete the object if it so chooses.

```
            case CLASSIFY_BEHIND:
                // Nothing was in front
                NewPoints = NULL;
                break;
```

We will see later when we cover the core clipping function of the PVS calculation how we sometimes want to clip away a polygon even if it is located on the plane. For example, if the generator portal is located on the same plane as the target portal then the target portal can not possibly see through the generator portal and should be completely clipped away.

You can see below that in the on plane case, if the KeepOnPlane Boolean parameter is set to true, we just assign the NewPoints pointer to the ‘this’ pointer allowing the object to return a pointer to itself. Otherwise, we break and NULL will be returned at the bottom of the function signifying to the caller that the portal should be deleted.

```
            case CLASSIFY_ONPLANE:
                // Should we keep the onplane case ?
                if ( KeepOnPlane ) NewPoints = this;
                break;
```

Finally, if the polygon is spanning the plane, we create a new CPortalPoints object which is passed into the Split function to retrieve the front fragment. NULL is passed as the back split as we do not wish to retrieve the back fragment of the polygon as this is the fragment that should be discarded.

```
            case CLASSIFY_SPANNING:

                // Allocate a new set of points
                NewPoints = new CPortalPoints;
                if (!NewPoints) throw std::bad_alloc();

                // Clip the current portal points
                if ( FAILED(Split( Plane, NewPoints, NULL ))) throw BCERR_OUTOFMEMORY;
                break;

        } // End Switch

    } // End try block

    catch (...)
    {
        return NULL;

    } // End catch block

    // Success!!
    return NewPoints;
}
```

The CProcessPVS Class

With the initial support structures covered, we will now look at the code to the CProcessPVS module. It is declared in ProcessPVS.h and is shown below. First four public methods of the object's interface should be familiar as they form the method set common to all our modules. They include the Process method that is called by CCompiler to invoke the PVS calculator module and the methods to set the modules Logger, parent and options. There are also two additional public methods called 'GetPVSPortalCount' and 'GetPVSPortal' which can be used to fetch the one-way portal information stored in the module. There are also many private functions which are used by the Process method to accomplish its task which we will examine in a moment.

```
class CProcessPVS
{
public:
    // Constructors & Destructors for This Class.
             CProcessPVS();
    virtual ~CProcessPVS();

    // Public Functions for This Class.
    HRESULT         Process( CBSPTree * pTree );
    void            SetOptions( const PVSOPTIONS& Options ) { m_OptionSet = Options; }
    void            SetLogger ( ILogger * pLogger )         { m_pLogger = pLogger; }
    void            SetParent ( CCompiler * pParent )       { m_pParent = pParent; }

    unsigned long   GetPVSPortalCount( ) const
                               { return (unsigned long)m_vpPVSPortals.size(); }

    CPVSPortal     *GetPVSPortal( unsigned long Index ) const
                               { return (Index < m_vpPVSPortals.size()) ?
                                         m_vpPVSPortals[Index] : NULL; }

private:
    // Private Functions for This Class.
    HRESULT         GeneratePVSPortals( );
    HRESULT         InitialPortalVis( );
    HRESULT         CalcPortalVis( );
    void            PortalFlood( CPVSPortal * SourcePortal,
                                 unsigned char PortalVis[],
                                 unsigned long Leaf );

    HRESULT         ExportPVS( CBSPTree * pTree );

    void            GetPortalPlane( const CPVSPortal * pPortal, CPlane3& Plane );

    ULONG           CompressLeafSet ( UCHAR MasterPVS[],
                                      const UCHAR VisArray[],
                                      ULONG WritePos);

    ULONG           GetNextPortal();

    HRESULT         RecursePVS( ULONG Leaf, CPVSPortal * SourcePortal, PVSDATA & PrevData );

    CPortalPoints * ClipToAntiPenumbra( CPortalPoints * Source,
                                        CPortalPoints * Target,
                                        CPortalPoints * Generator,
                                        bool ReverseClip );

    // Private Static Functions for This Class.
```

```
    static CPortalPoints *  AllocPortalPoints( const CPolygon * pPolygon, bool Duplicate );
    static bool             GetPVSBit( UCHAR VisArray[], ULONG DestLeaf );
    static void             SetPVSBit( UCHAR VisArray[], ULONG DestLeaf, bool Value = true );
    static void             FreePortalPoints( CPortalPoints * pPoints );

    // Private Variables for This Class.
    PVSOPTIONS      m_OptionSet;        // The option set for PVS Compilation.
    ILogger        *m_pLogger;          // Logging interface used to log progress etc.
    CCompiler      *m_pParent;          // Parent Compiler Pointer

    CBSPTree       *m_pTree;            // The tree used to compile the PVS.
    ULONG           m_PVSBytesPerSet;   // Number of Bytes required to
                                        // describe a single leaf's visibility
    vectorPVSPortal m_vpPVSPortals;     // Vector storage of pointers to CPVSPortal objects
};
```

We will first examine the member variables of this object before examining each of its methods.

PVSOPTIONS m_OptionSet;
This member holds the compilation options for the PVS module and is set by CCompiler via the CProcessPVS::SetOptions method. We looked at this structure earlier and saw that it contained members to instruct the module to either do a full or fast compile (where fast simply use the PossibleVis array and does not perform the anti-penumbra clipping procedure) and describes the number of anti-penumbra clip tests (1 to 4) that should be carried out for each Source/Generator portal combination encountered during the recursive clipping procedure.

ILogger *m_pLogger;
This member will point to the logging object whose interface will be used by this module to output status reports and compilation errors or warnings. This is set by the CCompiler object prior to the Process function being called via the CporcessPVS::SetLogger method.

CCompiler *m_pParent;
This member is set by the CCompiler object via the CProcessPVS::SetParent method. It stores a pointer to the CCompiler object that invoked it.

CBSPTree *m_pTree;
This member points to the BSP tree that is having its PVS data calculated and that contains the portal information generated by the previously discussed module. This pointer is set by CCompiler via the parameter to the CProcessPVS::Process method.

ULONG m_PVSBytesPerSet;
This member will be calculated by this module at the start of the Process method and will contain the number of bytes needed to store the visibility information for a single leaf. As each byte contains 8 bits a single byte will contain the information for 8 leaves.

vectorPVSPortal m_vpPVSPortals;
This is an STL vector will store CPVSPortal structures. It is in this vector that the pointers of all the one-way portals generated by this module will be stored.

Constructor - CProcessPVS

The constructor of this module simply initializes all members to zero or NULL.

```
CProcessPVS::CProcessPVS()
{
    // Reset / Clear all required values
    m_PVSBytesPerSet    = 0;
    m_pLogger           = NULL;
    m_pTree             = NULL;
    m_pParent           = NULL;
}
```

Destructor - CProcessPVS

The destructor of this object must release the one-way portals (CPVSPortal structures) that it allocated to complete its task and must also empty the vector that contained these pointers.

```
CProcessPVS::~CProcessPVS()
{
    ULONG i;

    // Clean up after ourselves
    for ( i = 0; i < GetPVSPortalCount(); i++ ) if ( GetPVSPortal(i) )
                                                    delete GetPVSPortal(i);

    // Clear Vectors
    m_vpPVSPortals.clear();
}
```

AllocPortalPoints - CProcessPVS

There will be many times throughout the PVS calculation process that we will need to allocate new CPortalPoints object. In keeping with the allocation strategy we have used for other modules, this function can be used to allocate a new object of this type. The function simply wraps the allocation call.

The function takes two parameters. The first is a pointer to the CPolygon object (or derived object) that has the geometry we would like this new CPortalPoints object to represent and the second parameter is a Boolean that specifies whether we would like the CPortalPoints structure to copy the polygon data into its own vertex array or simply alias it. These parameters are simply passed into the CPortalPoints copy constructor which we have already covered.

```
CPortalPoints * CProcessPVS::AllocPortalPoints( const CPolygon * pPolygon, bool Duplicate )
{
    CPortalPoints * NewPoints = NULL;

    try
    {
        // Attempt to allocate a new set of points
```

```
        NewPoints = new CPortalPoints( pPolygon, Duplicate );
        if (!NewPoints) throw std::bad_alloc();

    } // End try block

    catch (HRESULT)
    {
        // Constructor throws HRESULT
        if (NewPoints) delete NewPoints;
        return NULL;

    } // End Catch

    catch ( std::bad_alloc )
    {
        // Failed to allocate
        return NULL;

    } // End Catch

    // Success!!
    return NewPoints;
}
```

FreePortalPoints - CProcessPVS

This module also has a method that can be used to free a CPortalPoints structure. It takes a single parameter, a pointer to the CPortalPoints structure that is to be released.

```
void CProcessPVS::FreePortalPoints( CPortalPoints * pPoints )
{
    // Validate Parameters
    if (!pPoints) return;

    // We are only allowed to delete NON-Owned point sets
    if ( pPoints->OwnerPortal == NULL ) delete pPoints;
}
```

Notice that this method will only physically delete the CPortalPoints object if it is not owned by a parent portal. If it is owned by a portal then the portal should be responsible for its clean up inside the portals destructor.

SetPVSBit - CProcessPVS

There will be several times throughout the PVS compilation procedure when we will need to set or clear a bit in a given portal's visibility bit set (be that its ActualVis array or its PossibleVis array). This method is a utility method that allows us to pass a bit set, a leaf whose visibility bit is to be altered in that set and a Boolean describing whether that leaf's bit should be set or cleared in the passed bit set. The function wraps calculating the bit that needs to be set in the passed bit array and setting that bit accordingly.

```
void CProcessPVS::SetPVSBit( UCHAR VisArray[], ULONG DestLeaf, bool Value /* = true */ )
{
    // Set / remove bit depending on the value
    if ( Value == true )
    {
        VisArray[ DestLeaf >> 3 ] |=  (1 << ( DestLeaf & 7 ));
    }
    else
    {
        VisArray[ DestLeaf >> 3 ] &= ~(1 << ( DestLeaf & 7 ));

    } // End if Value
}
```

The bit shifting logic is explained in the text book so refer back if you are feeling a little rusty. Recall that DestLeaf>>3 just divides it by eight which tells us the byte in which this leaf's visibility bit resides. By ANDing the leaf index with 7 (binary 00000111) we also get the bit within that byte that needs to be set (0 through 7). Therefore, we can shift a value of one by this amount to create a byte that has only that bit set (or unset) and then OR it with the byte in the bit set to toggle that leaf's bit on or off.

GetPVSBit - CProcessPVS

We also have a function that uses the same bit shifting logic to retrieve the visibility status of a leaf in the passed bit set.

```
bool CProcessPVS::GetPVSBit( UCHAR VisArray[], ULONG DestLeaf )
{
    return (VisArray[ DestLeaf >> 3 ] & (1 << ( DestLeaf & 7))) != 0;
}
```

With these utility functions out of the way, let us now look at the function that makes it all happen. The CProcessPVS::Process method.

Process - CProcessPVS

This function is the parent function of the PVS calculation process that calls the various sub-processes in order to calculate the final PVS for the passed BSP tree. We will look at the code a section at a time.

The first thing we do in this function is test that the passed tree has portal data generated for it. If this is not the case we return immediately as we can not possibly calculate PVS data for this tree. Otherwise, we copy the passed BSP tree pointer into the module's member variable.

```
HRESULT CProcessPVS::Process( CBSPTree * pTree )
{
    HRESULT hRet;

    // Validate values
    if (!pTree) return BCERR_INVALIDPARAMS;

    // Validate Input Data
    if ( pTree->GetPortalCount() == 0 ) return BCERR_BSP_INVALIDTREEDATA;
```

```
    // Store tree for compilation
    m_pTree = pTree;
```

The next thing we do is calculate the module's m_PVSBytesPerSet member such that it describes the number of bytes needed to hold the visibility bit set for a single leaf. As discussed in the text book, although there are 8 bits per byte with each bit representing a leaf's visibility, we can not just divide the total leaf count of the tree by 8 to calculate this size. If we did it would erroneously calculate the number of bytes to be allocated to store a 9 leaf tree as 9/8 = 1 (integer math). As we know, we would actually need two bytes to represent this information with the first 8 leaves having their bits in byte one and the 9^{th} leaf with its bit in the first position in byte 2. Therefore, to cope with the integer truncation we add 7 to the leaf count first and then divide this by 8. Therefore, if we had a 9 leaf tree, the number of bytes we would need to represent a bit for each leaf would be (9+7)/8 = 16/8=2.

```
    // Calculate Number Of Bytes needed to store each leafs
    // vis array in BIT form (i.e 8 leafs vis per byte uncompressed)
    m_PVSBytesPerSet = (pTree->GetLeafCount() + 7) / 8;
```

For reasons that will become clear in the core clipping process we also wish to pad this size to the nearest four byte boundary. This will allow us to access the visibility byte arrays using a long pointer and iterate through 4 bytes (32 leaves) at a time. For example, imagine that we allocated three bytes and then tried to write to the first byte with a four-byte pointer (long *). In this case, we would accidentally write to the fourth byte, overstepping the bounds of the array and writing to invalid memory. Therefore, if we need 22 bytes, we will allocate 24 bytes instead; the two bytes at the end will serve as padding only and will never be used by us.

```
    // 32 bit align the bytes per set to allow for our early out long conversion
    m_PVSBytesPerSet = (m_PVSBytesPerSet * 3 + 3) & 0xFFFFFFFC;
```

This m_PVSBytesPerSet member now describes the size that we will need to allocate each portal's PossibleVis and ActualVis arrays later in the process. These arrays will be padded with extra unused bytes at the end of the array if necessary to make sure we can access the array four bytes at a time without overflowing the array. The remainder of the function calls four member functions and clearly shows the four sub-process involved in generating the PVS data for the tree.

First we call the GeneratePVSPortals method. This is the method that will fetch each two-way portal from the tree and will generate two one-way CPVSPortal structures from it. It is the CPVSPortals that will be used by the PVS calculator to control flow visibility as we recur through the level.

```
    // Retrieve all of our one way portals
    hRet = GeneratePVSPortals();
    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;
```

After the above function returns, this object's m_vpPVSPortals vector will be filled with all the one-way portals needed to perform PVS calculation. Because of their one-way nature, there will be twice as many CPVSPortals in this array as there are CBSPPortals stored in the BSP tree.

With the one-way portals generated, next we call the InitialPortalVis function. It is this function that will perform the initial flood fill through the level and calculate the PossibleVis array for each portal. You will recall that the PossibleVis array stored in each portal is a very approximate leaf visibility array for that portal which will be used to speed up the core clipping process when the portal's ActualVis array is calculated.

```
    // Calculate initial portal visibility
    hRet = InitialPortalVis();
    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;
```

When the above function returns each CPVSPortal will contain its PossibleVis array of approximate leaf visibility information. Next we enter the core clipping process with a call to CalcPortalVis. This function is the function that will calculate the ActualVis array for each portal. That is, the function that will generate the leaf visibility information for each portal, the portals' PVS.

```
    // Perform actual full PVS calculation
    hRet = CalcPortalVis();
    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;
```

At this point in the function the PVS for each portal will have been calculated so we next call the ExportPVS method. This method will use the portal PVS's to calculate the PVS for each leaf in the tree. Recall that a leaf's PVS is simply the accumulation of the PVS's of each portal residing in that leaf. After the PVS for each leaf has been calculated this information will be compressed and stored in the BSP tree in a single byte array. The leaves of the tree will also have their PVSIndex members pointing to the correct location in this array describing the starting location of the leaf's visibility information within the final PVS data block.

```
    // Export the visibility set to the final BSP Tree master array
    hRet = ExportPVS( pTree );
    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;

    // Success
    return BC_OK;
}
```

After the ExportPVS method returns, all that is left to do is return from the function as the PVS information has been calculated, optionally compressed using zero run length encoding and stored in the BSP tree. Our module at this point has completed its task and can return program flow back to CCompiler.

> **Note:** Recall that after the PVS module has returned after completing its process, CCompiler will then activate the T-junction repair module as a final step before saving the BSP tree out to disk in the form of an IWF file.

We will now cover the methods called from the above function in order and get a real understanding of where and when everything happens throughout the PVS calculation process. The first function to be called is the GeneratePVSPortals method so we will examine the code to this function first.

GeneratePVSPortals - CProcessPVS

This method has a very simple task. It has to loop through each CBSPPortal (two-way portal) stored in the BSP tree and generate two CPVSPortals (one-way portals) for it. Each one-way portal will have a different neighbor leaf selected from the two leaves in which the two-way portal resides. This will dictate the direction in which the one-way portals are assumed to facing.

The first section of the code fetches the number of portals stored in the BSP tree. We then set up a loop to allocate twice this many CPVSPortals and store their pointers in the CProcessPVS::m_vpPVSPortals vector.

```
HRESULT CProcessPVS::GeneratePVSPortals( )
{
    ULONG i, p, PortalCount = m_pTree->GetPortalCount();

    // Allocate enough PVS portals to store one-way copies.
    try
    {
        m_vpPVSPortals.resize( PortalCount * 2 );
        for ( i = 0; i < PortalCount * 2; i++ )
        {
            // Allocate a new portal
            m_vpPVSPortals[i] = new CPVSPortal;
            if (!m_vpPVSPortals[i]) throw std::bad_alloc();

        } // Next Portal

    } // Try vector ops

    // Catch Failures
    catch (...)
    {
        return BCERR_OUTOFMEMORY;

    } // End Catch
```

At this point we have allocated the correct number of one-way portals and have their pointers stored in the module's one-way portal vector. However, these portals are still un-initialized as we have not yet populated them with geometry.

In the next step we will loop through each of the portals we just calculated two at a time. We process the one-way portals in pairs in the following loop as each pair of one-way portals is a child (to use the term very loosely) of a single one-way portal in the BSP tree. So you can see that we set up the following loop to step through the one-way portal list two portals at a time. Heee is the first section of the loop code.

```
    // Loop through each portal, creating the duplicate points
    for ( i = 0, p = 0; i < PortalCount; i++, p+=2 )
    {
        // Retrieve BSP Portal for easy access
        CBSPPortal * pBSPPortal = m_pTree->GetPortal(i);
```

```
        CPortalPoints *pp = AllocPortalPoints( pBSPPortal, false );
        if ( !pp ) return BCERR_OUTOFMEMORY;
```

Inside the loop we fetch a pointer to the original two-way portal from the BSP tree whose geometry we are going to copy/alias using the pair of one-way portals we are currently processing. pBSPPortal points to the original portal in the BSP tree which we will use to set up two one-way portals. We then allocate a new CPortalPoints object. Remember, this is the polygon that will be stored in the two CPVSPortals we are about to populate.

Notice how we clone the CPortalPoints object from the BSP portal. Although we are creating two one way portals, we only need to create one CPortalPoints objects as each one-way portal will share the same physical polygon data. Only one of the CPVSPortals we are about to populate will actually own the CPortalPoints structure and will be responsible for deleting it within its destructor. The other CPVSPortal object will simply reference it.

Another important point to notice is how we also pass false into the copy constructor so that the CPortalPoints object does not have its own array of vertex data allocated, but simply aliases the vertex array stored in the original BSP portal. Therefore, we will create two CPVSPortal structures that share the same CCPortalPoints object and that CPortalPoints object will share its vertex data with the original BSP portal. It is clear then that the two CPVSPortals we are about to populate share the same vertex data, the vertex data in the original BSP polygon.

Here is the remainder of the loop (and the function) that sets up each CPVSPortal which we discuss beneath it.

```
        // Create link information for front facing portal
        m_vpPVSPortals[p]->Points        = pp;
        m_vpPVSPortals[p]->Side          = FRONT_OWNER;
        m_vpPVSPortals[p]->Status        = PS_NOTPROCESSED;
        m_vpPVSPortals[p]->Plane         = m_pTree->GetNode( pBSPPortal->OwnerNode )->Plane;
        m_vpPVSPortals[p]->NeighbourLeaf = pBSPPortal->LeafOwner[ FRONT_OWNER ];
        m_vpPVSPortals[p]->OwnsPoints    = true;

        // Store owner portal information (used later)
        pp->OwnerPortal = m_vpPVSPortals[p];

        // Create link information for back facing portal
        m_vpPVSPortals[p + 1]->Points   = pp;
        m_vpPVSPortals[p + 1]->Side     = BACK_OWNER;
        m_vpPVSPortals[p + 1]->Status   = PS_NOTPROCESSED;
        m_vpPVSPortals[p + 1]->Plane    = m_pTree->GetNode( pBSPPortal->OwnerNode )->Plane;

        m_vpPVSPortals[p + 1]->NeighbourLeaf = pBSPPortal->LeafOwner[ BACK_OWNER ];
        m_vpPVSPortals[p + 1]->OwnsPoints    = false;

    } // Next Portal

    // Success!!
    return BC_OK;
}
```

Notice how each one-way portal we populate points to the same CPortalPoints structure but only the first portal we create has its OwnsPoints Boolean set to true. The first one-way portal is therefore the owner of the polygon itself and is responsible for the cleanup of that polygon in its destructor.

Notice also that because we know that the BSP portal has the index of the leaf in its front space stored in element zero (FRONT_OWNER) in its LeafOwner array and the leaf that is located in its back space located in element 1 (BACK_OWNER) in the portal's LeafOwner array, we can easily set up portals that correctly point into the relevant neighbor leaves. For example, the first portal we set up is assigned the Side value of front owner which means we intend this portal to be the one that faces in the same direction as the owner node's plane. When this is the case we know that its neighbor leaf (the leaf its normal should point into should a normal actually exist) is in the front space of the node plane and as such, we assign to its NeighborLeaf member the leaf index stored in the BSP portal's LeafOwner array at element FRONT_OWNER. Notice that we simply flip this logic to set up the 2nd one-way portal so that its side is set to BACK_OWNER. This means its normal is assumed to flow into the back space of the leaf and thus, it faces in the opposite direction to that of the original node plane.

Finally, notice how we also store the plane of the portal's owner node (the portals plane) in the CPVSPortals. As discussed earlier, although the portals are supposed to be facing in opposing directions we store the same clip plane in both. The Side member will instruct the core clipping process to flip the plane orientation of the back facing portal prior to clipping anything against it. This may seem a little unclear at the moment which is why we will jump ahead temporarily and look at a function that will be used later during the main clipping process, the GetPortalPlane function.

GetPortalPlane - CProcessPVS

This function is used by the PVS calculator when it wishes to retrieve the plane of a CPVSPortal object. This is often required when generator portals need to be clipped against the plane of the source portal and vice versa. As our clip functions will always remove portions of a polygon/portal that lay in the back space of a clip plane, and considering that the two one-way way portals that exist on the same plane and were duplicated from the same two way polygon both store the same plane, it is obvious that just using this clip plane for the back facing portal (Side = BACK_OWNER) would cause errors. That is, as the plane normal is facing into the opposite half space as the portal, clipping anything away that lies behind this plane would actually clip away anything that lay in the front half space of the portal causing obvious PVS errors. To fix this problem the GetPortalPlane method is used to retrieve the plane of a portal. If the portal whose plane is being retrieved has its side set to BACK_OWNER, then the plane orientation is flipped prior to it being returned. This will make sure that the returned plane always faces into the same front space as the portal flows.

```
void CProcessPVS::GetPortalPlane( const CPVSPortal * pPortal, CPlane3& Plane )
{

    // Store plane information
    Plane = *m_pTree->GetPlane( pPortal->Plane );

    // Swap sides if necessary
    if ( pPortal->Side == BACK_OWNER )
    {
        Plane.Normal   = -Plane.Normal;
```

```
        Plane.Distance = -Plane.Distance;

    } // End if Swap Sides
}
```

InitialPortalVis - CProcessPVS

After the one-way portal array has been created, the Process method next calls the InitialPortalVis method. It is this method that performs the flood fill through the leaves of the tree and constructs a very crude visibility array (PossibleVis) for each portal. As we have discussed, this array will be used to optimize the main PVS calculation procedure (the anti-penumbra clipping process).

The function first sends some information to the PVS logging channel specifying that the initial portal flow is about to be calculated and the progress range of the logger is set to the total number of portals in the one-way portal array. Once we have processed each portal in this array and calculated its PossibleVis array, we will have completed the initial portal flow procedure.

```
HRESULT CProcessPVS::InitialPortalVis()
{
    CPortalPoints  *pp;
    ULONG           p1, p2, i;
    CPlane3         Plane1, Plane2;
    UCHAR          *PortalVis = NULL;
    CPVSPortal     *pPortal1, *pPortal2;

    // *************************
    // * Write Log Information *
    // *************************
    if ( m_pLogger )
    {
        m_pLogger->LogWrite( LOG_PVS,
                             0,
                             true,
                             _T("Calculating initial PVS portal flow \t\t- " ) );

        m_pLogger->SetRewindMarker( LOG_PVS );
        m_pLogger->LogWrite( LOG_PVS, 0, false, _T("0%%" ) );
        m_pLogger->SetProgressRange( GetPVSPortalCount() );
        m_pLogger->SetProgressValue( 0 );
    }
    // ************************
    // *    End of Logging    *
    // ************************
```

The rest of the function is essentially just a loop through each portal that performs a flood fill and is comprised of two different stages.

In the first stage we calculate which portals in the level could conceivably have portal flow with the current portal being processed. This allows us to compile a temporary byte array large enough to store a byte for each portal. Each byte in this array is set to either 1 or 0 depending on whether that portal can be seen from the current portal being processed. Portals that have their bytes set to zero in this array will essentially stop the flood for this portal. Once we have this portal visibility array compiled for the current portal being processed, it will be passed into a recursive flood filling function and used to

calculate the PossibleVis array for the current portal. Remember, the PossibleVis array of the portal is a bit set containing the 'possible' leaf visibility array of the portal. That is, which leaves in the tree have the potential to be visible during the anti-penumbra clipping process.

So, the first thing we must construct inside the loop that iterates through each portal, is the array of bytes that describes which portals can be seen by the current portal being processed.

We loop through each of the other portals and perform two tests on it to determine whether its corresponding byte in the array should be set to zero or one for the current source portal.

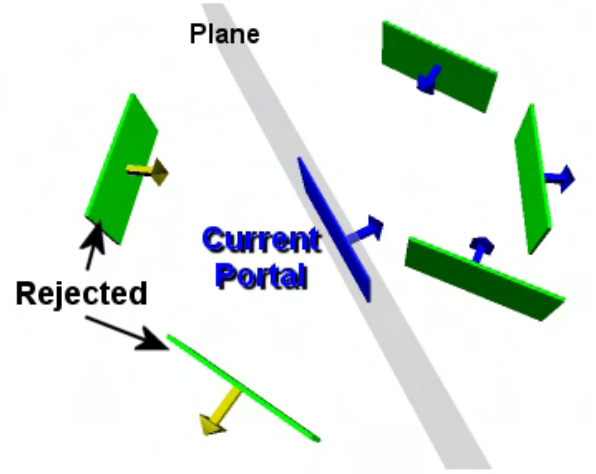


Figure 17.3

Firstly, as portal flow passes through the back of a portal out into its neighbor leaf, we know that any portals that are located behind the plane of the current portal being processed can not possibly be visible to the current portal as shown in figure 17.3. We can see that in the initial test we find two portals located behind the current portal's plane. This means visibility flow can not exist out of the front of the current portal and through the back of these two portals. That is, the current portal can only see into its front space and therefore can not possible see through portals in its back space. As such, the two portals in the example would have their bytes set to zero in the current portals array.

As flow can only exist out of the front of one portal and through the back of another, we can also see that for portals in the front space of the current portal which face towards the current portal no flow can exist also.

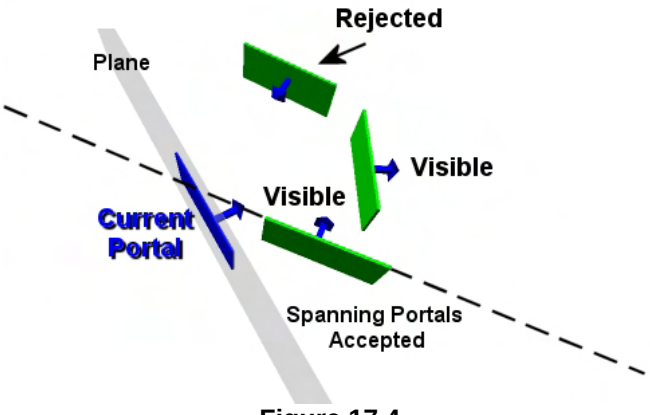


Figure 17.4

In figure 17.4 we can see that in this second test another portal is also rejected by the visibility test because its faces into the opposing half space to the current portal. That is, the text portal and the current portal are facing each other.

We can see that the middle portal on the right hand side of the diagram is not rejected from the visibility test because it faces into the same half space as the current portal. Therefore, we can clearly see that the back side of this portal is clearly visible from the current portal and as such, portal flow can happen between these two portals. The bottom portal of the three test portals is also accepted even though its plane spans the current portal. We will have to sort this problem out later in the main recursive process and clip such portals to each others plane, but for now it is accepted as at least some of the current portal lay in the test portals back space and can see through the back of that portal. Therefore, portal flow does occur between these portals to some degree.

Let is now see the code that performs these tests and compiles the temporary portal/portal visibility buffer. This should obviously be large enough to store a byte for each portal.

```
    try
    {
        // Allocate temporary visibility buffer
        if ( !(PortalVis = new UCHAR[ GetPVSPortalCount() ])) throw std::bad_alloc();

        // Loop through the portal array allocating and checking
        // portal visibility against every other portal
        for ( p1 = 0; p1 < GetPVSPortalCount(); p1++)
        {
            // Update progress
            if (!m_pParent->TestCompilerState()) break;
            if ( m_pLogger ) m_pLogger->UpdateProgress( );

            // Retrieve first portal for easy access
            pPortal1 = GetPVSPortal( p1 );

            // Retrieve portal's plane
            GetPortalPlane( pPortal1, Plane1 );

            // Allocate memory for portal visibility info
            if (!(pPortal1->PossibleVis = new UCHAR[m_PVSBytesPerSet]))
                  throw std::bad_alloc();

            ZeroMemory( pPortal1->PossibleVis, m_PVSBytesPerSet );

            // Clear temporary buffer
            ZeroMemory( PortalVis, GetPVSPortalCount() );
```

Starting at the top of the above section of code we can see that we allocate the byte array that will be temporarily used by each portal we process to contain its portal/portal visibility information. We allocate it large enough to store a byte for each one-way portal.

We then set up a loop to iterate through each portal and compute its possible visibility. After updating the logger's progress we fetch the pointer to the portal that is to be process p1. We then fetch the portals plane using the GetPortalPlane method. Recall from earlier that this function will take care of returning a plane which always faces into the portals neighbor leaf. As this portal (pPortal1) is about to have its PossibleVis array calculated we allocate this array next. Notice that it is allocated to m_PVSBytesPerSet in size. The value of this variable was calculated at the beginning of the Process method and contains how many bytes (including 4 byte alignment padding) we must allocate to have an array where we can represent a single visibility bit for each leaf in the tree. We then zero this array initially so that all leaves are considered invisible to this portal by default. We also zero the PortalVis buffer which is the buffer we will use temporarily in this loop to calculate the portal/portal visibility information (byte for each portal).

Now that we have all the information for the current portal that we are processing, we will set up a loop to iterate through all the other portals in the scene and will perform the two visibility tests illustrated in figures 17.3 and 17.4. At the head of this inner loop we obviously skip the tests if the test portal is equal to the current portal being processed. If this is not the case however, then we fetch a pointer to the test portal and its plane.

```
            // For this portal, loop through all other portals
            for ( p2 = 0; p2 < GetPVSPortalCount(); p2++)
            {
                // Don't test against self
                if (p2 == p1) continue;

                // Retrieve second portal for easy access
                pPortal2 = GetPVSPortal( p2 );

                // Retrieve portal's plane
                GetPortalPlane( pPortal2, Plane2 );
```

For the first test we loop though each of the test portals vertices and classify their positions against the plane of the current portal. If any vertex is found to exist in the front space of the current portal then some portal flow may exist so we break. If we do not break from the loop prematurely however, then it means we must have found that the test portal is contained completely in the back space of the current portal's plane. When this is the case the loop variable 'i' will equal the vertex count of the test portal on loop exit which we can test and skip any further processing if this is the case. That is, if the test portal is found to be in the back space of the current portal then it can not possibly be visible and no portal flow can exist between them so we skip any further tests and never set its byte to one in the PortalVis array.

```
                // Test to see if any of p2's points are in front of p1's plane
                pp = pPortal2->Points;
                for ( i = 0; i < pp->VertexCount; i++)
                {
                   if ( Plane1.ClassifyPoint( pp->Vertices[i] ) == CLASSIFY_INFRONT ) break;

                } // Next Portal Vertex

                // If the loop reached the end, there were no points in front so continue
                if ( i == pp->VertexCount ) continue;
```

If we get this far then it means the test portal (or a fragment of it) must be located in front of the current portal. Our next test is to see if the portals are facing into each other because if this is the case, no portal flow can exist between them also.

As the portals do not contain normals we perform this test by classifying each vertex in the current portal against the plane of the test portal. If portal flow exists between these portals then some of the vertices of the current portal must be located in the back space of the test portal. As soon as we find a vertex that is contained in the back space of the test portal we break from the loop.

```
                // Test to see if any of p1's portal points are Behind p2's plane.
                pp = pPortal1->Points;

                for ( i = 0; i < pp->VertexCount; i++)
                {
                    if ( Plane2.ClassifyPoint( pp->Vertices[i] ) == CLASSIFY_BEHIND ) break;

                } // Next Portal Vertex

                // If the loop reached the end, there were no points in front so continue
                if ( i == pp->VertexCount ) continue;
```

If the loop does not exit prematurely then it means the current portal is in the front space of the test portal and therefore, these portals must be facing each other. When this is the case, loop variable ‘i’ will be equal to the current portal’s vertex count so we can perform this comparison and skip this portal if this is the case. That is, the back of the test portal is not visible to the current portal so no portal flow can exist between them.

If the test portal passes both these tests then it means portal flow does exist between the current portal and the test portal so we set the test portal’s byte to 1 in the PortalVis buffer.

```
                // Fill out the temporary portal visibility array
                PortalVis[p2] = 1;

            } // Next Portal 2
```

After the above inner loop has completed, PortalVis will contain a byte set to 1 for every portal that is potentially visible from the current portal. All we have to do now is perform the flood fill using this information.

```
            // Now flood through all the portals which are visible
            // from the source portal through into the neighbour leaf
            // and flag any leaves which are visible (the leaves which
            // remain set to 0 can never possibly be seen from this portal)
            pPortal1->PossibleVisCount = 0;
            PortalFlood( pPortal1, PortalVis, pPortal1->NeighbourLeaf );

        } // Next Portal
```

As you can see, before performing the flood fill we set the current portals PossibleVisCount to zero as we have not yet found any leaves to be visible, this is what the PortalFlood method will determine. We then call the PortalFlood function (which we will discuss next) to perform the flood. We pass in the current portal which is to have its PossibleVis array calculated, the PortalVis buffer which describes where the flood is blocked by non-visible portals and as the third parameter we pass in the neighbor leaf index of the current portal. It is in this leaf the portal flood will begin. When the outer portal loop exits, every portal will have had a chance to be the current portal and will have had its PossibleVis array calculated using the PortalFlood function.

Before we continue we next test that the compiler has not had a notification to cancel the compilation and if it has received such a notification we return. (Remember that the compiler is running in its own thread).

```
        // If we're cancelled, clean up and return
        if ( m_pParent->GetCompileStatus() == CS_CANCELLED )
        {
            if ( PortalVis) delete[]PortalVis;
            return BC_CANCELLED;

        } // End if Cancelled.

    } // End Try Block

    // Catch Bad Allocations
```

```
    catch ( std::bad_alloc )
    {
        // Clean up and return (Failure)
        if (PortalVis) delete []PortalVis;
        if ( m_pLogger ) m_pLogger->ProgressFailure( LOG_PVS );
        return BCERR_OUTOFMEMORY;

    } // End Catch Block
```

At the bottom of the function we delete the temporary portal/portal visibility buffer and inform the logger that the process has been a success before returning from the function.

```
    // Clean up
    if (PortalVis) delete []PortalVis;

    // Success!!
    if ( m_pLogger ) m_pLogger->ProgressSuccess( LOG_PVS );
    return BC_OK;
}
```

As was evident from look at the above code, a call is made to the PortalFlood function for each portal it processes to generate that portals 'possible' visibility set. Let us have a look at that function next.

PortalFlood - CProcessPVS

This recursive function is passed a source portal which needs to have its PossibleVis array populated with leaf visibility information. It is also passed a byte array describing, for the passed portal, which other portals in the level are visible. As the third parameter the neighbor leaf of the portal is passed. This is the leaf that is located immediately in front of the passed portal and is the leaf at which the flood fill for the portal will begin. This function will start from the neighbor leaf and will recursively flood out through its portals into neighboring leaves. The flood stops in any given direction when we enter an area where its portals are no longer considered visible from the passed source portal. That is, when we reach a portal that has its corresponding byte in the PortalVis buffer set to zero.

For each leaf we recur into via a portal we set its visibility bit in the source portals PossibleVis array to 1. Figure 17.5 demonstrates this process.

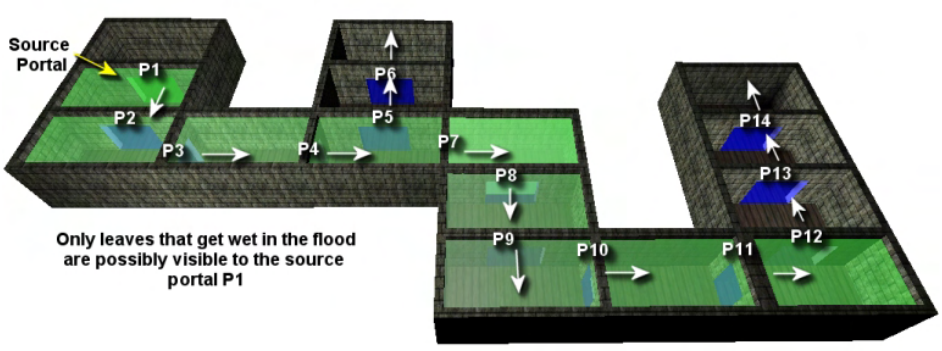


Figure 17.5

We can see that the flood starts in the neighbor leaf of portal P1 and flows through every portal for which portal flow exists and marks that portal's neighbor leaf as visible. The pattern repeats until we run out of visible portals to flood through. You can see in figure 17.5 that we do not flood through portals P5 or P2 as these portals were found to have no portal flow with the source portal (P1) in the previous method. That is, we only flow through a portal and mark its neighbor leaf as visible if its corresponding

byte in the passed PortalVis array is set to 1. In figure 17.5 the leaves that are highlighted green were marked as possibly visible by this flood filling process.

In the first section of the function we use the passed neighbor leaf index to fetch the leaf structure from the BSP tree's leaf array. As this is the leaf we are currently visiting it means the flood has made it into this leaf and therefore this leaf must be visible from the portal in question. However, because we do not wish to ever get stuck in a recursive loop we must make sure that we have not visited this leaf before in the flood so that we do not flow through its portals again. You can see that we use the GetPVSBit method to fetch the visibility bit in the portals PossibleVis array for the current leaf and if it is already set to 1, it means we have already visited this leaf before and have marked it as visible. When this is the case we simply return. If this is not the case however, we use the SetPVSBit method to mark the current leaf as visible in the portals PossibleVis array.

```
void CProcessPVS::PortalFlood( CPVSPortal * SourcePortal,
                               unsigned char PortalVis[],
                               unsigned long Leaf )
{
    CBSPLeaf * pLeaf = m_pTree->GetLeaf( Leaf );

    // Test the source portals 'Possible Visibility' list
    // to see if this leaf has already been set.
    if ( GetPVSBit( SourcePortal->PossibleVis, Leaf ) ) return;

    // Set the possible visibility bit for this leaf
    SetPVSBit( SourcePortal->PossibleVis, Leaf );

    // Increase portals 'Complexity' level
    SourcePortal->PossibleVisCount++;
```

Notice in the above code that after we have marked the leaf as visible to the portal we increment the portal's PossibleVisCount so that when the flood is complete, this member will contain all the visible leaves that were found. That is, the number of leaves that got wet by the flood fill. As described earlier, this PossibleVisCount member will be used as a measure of the portal's potential complexity and will be used by the core clipping process to determine which portal should have its PVS calculated next. Portals with a smaller PossibleVisCount will be selected for PVS calculation first.

Now that we have identified that the current leaf is visible it is now time to loop through all the portals stored in that leaf. If the portal is visible to the current source portal we will recur through that portal into its neighbor leaf. Otherwise we will skip the portal. Here is the remainder of the function code.

```
    // Loop through all portals in this leaf (remember the portal numbering
    // in the leaves match up with the originals, not our PVS portals )
    for ( ULONG i = 0; i < pLeaf->PortalIndices.size(); i++)
    {
        // Find correct portal index (the one IN this leaf (not Neighbouring))
        ULONG PortalIndex = pLeaf->PortalIndices[ i ] * 2;
        if ( GetPVSPortal( PortalIndex )->NeighbourLeaf == Leaf ) PortalIndex++;

        // If we can't pass through this portal then continue to next portal
        if ( !PortalVis[ PortalIndex ] ) continue;

        // Flood fill out through this portal
        PortalFlood( SourcePortal, PortalVis, GetPVSPortal( PortalIndex )->NeighbourLeaf );
```

```
    } // Next Leaf Portal
}
```

Notice that because we are accessing the portal information from the leaf of the BSP tree which stores one two-way portal instead of a pair of one-way portals, we need to multiply the leaf's portal index by 2 to get the index into the one-way portal array of the first in the pair of one-way portals that were generated for it. If the first one-way portal in the pair has a neighbor leaf equal to the leaf we are already in then this is obviously the one-way portal that points back into this leaf. This is obviously not the one-way portal that flows out of the neighbor leaf and therefore, the portal we are after must be the second in the pair so we increment the leaf index so that it addresses the correct one-way portal. Next we test to make sure this portal is visible by testing its corresponding byte in the PortalVis array, and if it is visible, the function calls itself to recur into the neighbor leaf of this portal.

We have now covered two of the processes that are invoked from the main Process method of this module. At this point, program flow will have been returned back to the Process method and not only will the one-way portals have been created and stored, each of these portals will contain a PossibleVis array describing a very approximate leaf PVS for that portal. The next method invoked from the Process method is the CalcPortalVis function. This is the function that is the doorway to the core PVS calculation process and the function that kick starts the anti-penumbra clipping function that ultimately calculates the ActualVis array (the real PVS) for each portal.

The CalcPortalVis - CProcessPVS

Before we look at the CalcPortalVis method we must look at a structure that will be used to pass data from one recursion of the function to the next when performing the recursive clipping process. Because we will need to recur through the RecursePVS function many times (covered in a moment), it will be helpful to have a data structure where we can pass information from the previous recursion to the next. For example, when the RecursePVS function calls itself, the current generator portal needs to be passed into the next recursion as the target portal. We also need to be able to access the source portal at all times, regardless of what instance of the function we are in, so that we can set its visibility bits for each visible leaf we encounter. The structure is shown below.

Excerpt from CProcessPVS.h

```
typedef struct _PVSDATA                 // Structure to hold pvs processing data
{
        CPortalPoints   *SourcePoints;   // Current source portals points
        CPortalPoints   *TargetPoints;   // Current target portals points
        CPlane3          TargetPlane;    // The target's plane
        UCHAR           *VisBits;        // Visible Bits being calculated
} PVSDATA;
```

This structure is the transport mechanism with which to pass data about the generator portal that was selected in a previous iteration of the recursive function onto the next recursion. Since the previous generator portal becomes the target portal in the next recursion and is used to build a new anti-penumbra, this allows us to find a generator portal, fill out the information about it in one of these structures, and then pass on this structure to the next recursion so that we can access and use it as the

target portal during anti-penumbra generation. Let is discuss what the four members will be used for by the core PVS calculation process.

CPortalPoints *SourcePoints;
As we discuss in the text book, during the recursive clipping process the source portal will be the portal that is currently having is PVS calculated (ActualVis array populated). As we step recursively through generator portal after generator portal into new neighbor leaves, we need access to the source portals polygon data so that it can be used to create the anti-penumbra with the target portal at each step. When the RecursePVS function is first called, this will be a pointer to the source portal's polygon data.

However, we also discussed in the text book how at each step we build anti-penumbras between not only the source and target portals (allowing us to cull/clip generator portals) but also between the generator portal and the target portal which is used to clip a temporary copy of the source portal's polygon. By clipping the source portal and the generator portal at each step as small as possible, we assure that in the next recursive step a smaller anti-penumbra will be built allowing us to cull more generator portals from having to be visited. During the recursive process, this member will contain a copy of the source portal in its currently clipped state. That is, when at a given step in the process the source portal is clipped to the anti-penumbra and a new smaller source portal polygon is generated, that polygon is stored in this member so that it can be passed into the next recursion as the source portal geometry.

This is always a temporary copy of the source portal polygon from the first point it gets clipped. That is, although we set this to the CPortalPoints member of the source portal at the start of the process, we never want to clip or delete the original source portal as this will be needed for the calculation of other portals later in the process. Our hope is that as we step from leaf to leaf, the portal stored in this member will become smaller and smaller due to the ongoing clipping in each recursion until it generates a very small anti-penumbra in which no generator portals are contained and stopping the portal flow along that given path.

CPortalPoints *TargetPoints;
This member will be used to pass the geometry of the generator portal that has been stepped though in one recursion onto the next recursion where it will become the target portal and used to build the anti-penumbra planes.

As was the case with the source portal transport mechanism described above, a generator portal selected in one recursion of the function will be clipped to the current anti-penumbra in an attempt to make that generator as small as possible (or cull it completely) for the next recursion. The clipped copy of the generator portal is then stored in this member variable and passed into the next recursion of the function where it will be used as the target portal. By making both the source and target portals smaller and smaller with each generator portal that we step through, we will get a tighter and more accurate PVS calculated and raise the chances of being able to abort a from given path of portals earlier.

When the RecursePVS method is first called, this will be set to NULL as no target portal will have yet been selected. This allows us to identify the special case in the RecursePVS function where we have entered it for the very first time and we do yet have two portals with which to construct an anti-penumbra. In this first special case, no anti-penumbra clipping is performed. A generator portal is

simply selected and stored in the TargetPoints member. We then recur through that portal into its neighbor leaf where we enter the clipping process proper.

CPlane3 TargetPlane;
This will be used to store the plane of the target portal when input into the RecursePVS function. That is, this will be the plane of the generator portal that was selected in the previous recursion and the plane of the portal polygon stored in the TargetPoints member. As discussed in the accompanying text book, we will need this as we will need to clip the generator portal to the target portal's plane to make sure that a spanning case does not occur that could corrupt the anti-penumbra in the next recursion. The source portal will also be clipped to the generator portal's plane so all spanning case are eliminated. Remember, the only portion of the generator portal that should be visible is the portion that is contained in the front space of the source/target portal and vice versa.

The first time the RecursePVS function is called this will be set to the plane of the source portal. As we described above, the first call is a special case where a target portal has not yet been selected. This will allow us to make sure that the generator portal we selected (which will become the first target portal in the next recursion) will be clipped to the plane of the source portal as once again, the only portion of the target portal that should be visible to the source portal is the section that is contained in the source portal's front space.

UCHAR *VisBits;
This array will be used to accumulate and pass into the current recursion all leaves that can possibly be seen by the source portal given the current combination of portals we have stepped through up to this point. This allows us to very efficiently determine if there are any leaves to process down this path which might be visible to the source portal and if not, we can terminate this path of flow immediately. We know for example that at the start of the process each portal will have a bitset (PossibleVis) describing which leaves may be visible. However, we also know that this set can be refined by ANDing the bitsets of all portals we have stepped through up until this point. For example, imagine that in a 10 leaf level the PossibleVis array of the source portal has the following leaf bits set to 1.

Source Portal Vis - 1111100000

We can see at the start that only the first five leaves of the level could possibly be visible to the source portal as this was determined by the portal flood we performed earlier. Let us also assume that we next choose a target portal with which to build the anti-penumbra which had the following PossibleVis array calculated for it during the portal flood.

Target Portal Vis - 1110011100

We can see that the source portal has the possibility of being able to see into leaves 1 through 5 whilst the target portal has the possibility of seeing into leaves 1 through 3 and 6 through 8. Therefore, when we build an anti-penumbra between these two portals we are actually asking the question, "How much can we see from the source portal through the target portal" which at an approximate level can be determined by ANDing the visibility sets

Current Vis - 1111100000 + 1110011100 = 1110000000

As you can see, before we even perform any clipping we can tell that there can only possibly be 3 leaves that we need to visit and can be seen through the combination of portals we are currently looking through, leaves 1 through 3. It doesn't stop there though.

For example, imagine that we next choose a generator portal that has the following PossibleVis array

Generator Vis - 0011111111

When we combine its visibility set with the current visibility information we have collected so far we get:-

Current Vis - 111000000
AND
Generator Vis - 001111111

New Current Vis - 001000000

As you can see, this tells us that there is only one leaf we are interested in visiting at this point in the process. That is, there is only one leaf that is visible from the source portal when looking through the target and generator portals. This process is ongoing. As we step from recursion to recursion we combine the possible visibility sets of each portal we step through so we can very quickly determine whether there are any leaves to visit down this current path of portals through which we have traveled. This allows us to bail from the path early instead of having to perform the anti-penumbra clipping for generator portals which we have already determined have neighbor leaves that could not possibly be visible through the combination of portals we have traveled to get to this point.

The VisBits member of this structure allows us to pass this combined bitset into the next recursion where it will be combined with the next generator portals PossbleVis array and so on. That is, it inputs into the RecursePVS function a bitset that describes what leaves could possibly be visible given the portals we have traveled through to that point. When the RecursePVS function is first called, this will point to the source portal's PossibleVis array as we have not yet selected any target portals. For all future recursions this will contain the combination (the bitwise ANDing) of the source portals PossiblVis array with the PossibleVis arrays of all the portals we have walked through to get to the current leaf.

With this structure explained we can now look at the code to the CalcPortalVis function a section at a time and see how this structure is initially setup and passed into the first instance of the RecursePVS function.

In the first section of the code we test the compiler options for this module. If the option has not be set to enable a full PVS compile then it means the user does not wish us to perform the core time consuming anti-penumbra clipping tests at all and as such we should just use the PossibleVis arrays calculated for each portal in the portal flood fill as the actual PVS for each portal and return.

```
HRESULT CProcessPVS::CalcPortalVis()
{
    ULONG   i;
    HRESULT hRet;
    PVSDATA PVSData;

    // If we want to perform a quick vis (not at all accurate) we can
    // simply use the possible vis bits array as our pvs bytes.
     if ( !m_OptionSet.FullCompile )
     {
        for ( i = 0; i < GetPVSPortalCount(); i++ )
        {
            CPVSPortal * pPortal = GetPVSPortal( i );
             pPortal->ActualVis = pPortal->PossibleVis;

        } // Next Portal

        // We are finished here
        return BC_OK;

    } // End if !FullCompile
```

As you can see in the above code, if a full compile is not required we just loop through each portal and copy its PossibleVis array into its ActualVis array. As we know, the ActualVis array is where our compiler will expect the real PVS data of the portal to exist after the core clipping process has been performed. As we do not wish to perform this core clipping process we simply copy over the PossibleVis arrays into the ActualVis arrays of each portal so that the bitset is stored in the portal where the PVS exporter will expect to find it. We then return because our job is already done.

> **Note**: This option is not to be used for commercial output. The PVS set generated using such an option will be extremely over generous and run time application performance will suffer. This option is really for the developers of the project to perform a very quick compile to test the various assets of their game without having to wait long periods for PVS calculation every time they update the geometry of their level.

Now it is time to start the core PVS calculation process and as such we output to the logging device that PVS calculation is about to being. As the PVS process is essentially one of just looping through each portal and calculating its ActualVis array (that made it sound deceptively easy) we set the range of the logger to the portal count so that it can be increased with each portal we process.

```
    // *************************
    // * Write Log Information *
    // *************************
    if ( m_pLogger )
    {
        m_pLogger->LogWrite( LOG_PVS, 0, true, _T("Full PVS compile progress \t\t\t- " ) );
        m_pLogger->SetRewindMarker( LOG_PVS );
        m_pLogger->LogWrite( LOG_PVS, 0, false, _T("0%%" ) );
        m_pLogger->SetProgressRange( GetPVSPortalCount() );
        m_pLogger->SetProgressValue( 0 );
    }
    // ************************
    // *    End of Logging     *
    // ************************
```

Next we make sure that the PVSData structure that we pass into the initial call to the RecursePVS function is initialized to zero and then we set up a loop to iterate through each portal.

```
    try
    {
        // Clear out our PVSData struct
        ZeroMemory( &PVSData, sizeof(PVSDATA) );

        // Lets process those portal bad boys!! ;)
        for ( i = 0; i != -1; i = GetNextPortal() )
        {
            CPVSPortal * pPortal = GetPVSPortal( i );

            // Update Progress
            if (!m_pParent->TestCompilerState()) throw BC_CANCELLED;
            if (m_pLogger) m_pLogger->UpdateProgress();
```

Notice that we use the GetNextPortal method to choose the next portal in the list to have its PVS calculated. We will have a look at this function in a moment but as discussed earlier, we wish to process least complex portals first so this function simply returns the index of a portal that has not yet been processed and has the smallest number of visible leaves in its PossibleVis array. We then use this index to fetch the next portal to be processed (pPortal) which will become the source portal in the next call to the RecursePVS function. We also update the logger's progress as a new portal is about to have its PVS calculated.

Our next task is to populate the members of the PVSData structure that will be passed into the first recursion of the function. As explained above, we should initially set the SourcePoints member to point to the source portal's polygon geometry (its CPortalPoints structure) and should set the VisBits pointer to point at the PossibleVis array of the source portal. The Plane of the source portal is also retrieved and stored in the TargetPlane member. Although this last step seems a little strange, by storing the source portal's plane in the TargetPlane member for the first call to the RecursePVS function, it means the target portal that we do select in that function will be clipped to the source portals plane so that any spanning cases are eliminated.

```
            // Fill our our initial data structure
            PVSData.SourcePoints    = pPortal->Points;
            PVSData.VisBits         = pPortal->PossibleVis;
            GetPortalPlane( pPortal, PVSData.TargetPlane );
```

Notice that we do not set the TargetPoints member of the PVSData structure in the above section of code. This is left at NULL so that we can detect in the RecursePVS function that it is the first recur of the function and as such, no anti-penumbra has to be created. We can just choose a generator portal and recur into its neighbor leaf.

Because the current portal we are processing is about to have its ActualVis array calculated, we had better allocate the memory for it now so that it is large enough to store a bitset that represents the visibility information for each leaf and we should initially set all the bits in this array to zero so that they are all initially invisible. It will be the RecursePVS function that will set the bits to 1 in this array when it finds a leaf that is visible.

```
            // Allocate the portals actual visibility array
            pPortal->ActualVis = new UCHAR[ m_PVSBytesPerSet ];
            if (!pPortal->ActualVis) throw std::bad_alloc(); // VC++ Compat

            // Set initial visibility to off for all leaves
            ZeroMemory( pPortal->ActualVis, m_PVSBytesPerSet );
```

Now we call the RecursePVS function to calculate the PVS for this portal. We pass in the neighbor leaf index as the first parameter as this is the leaf for which the source portal first leads into and is where the recursive procedure should begin for this portal. As the second parameter we pass in the source portal itself. This will be needed by the RecursePVS function so that it can set bits to 1 in its ActualVis array each time it locates a visible leaf. As the final parameter we pass in the PVSData structure. When this function returns, the PVS for the portal will be stored in its ActualVis array so we set the status of this portal to PS_PROCESSED. This will assure that the GetNextPortal method used to select the next portal to be processed in the loop will not select this portal again.

```
            // Step in and begin processing this portal
            hRet = RecursePVS( pPortal->NeighbourLeaf, pPortal, PVSData );
            if ( FAILED( hRet ) ) throw hRet;

            // We've finished processing this portal
            pPortal->Status = PS_PROCESSED;

        } // Next Portal

    } // End Try Block
```

At this point the PVS for each portal has been calculated and is stored in their ActualVis arrays. All that is let to do is inform the logger to output success before returning from the function.

```
    // Catch all failures
    catch (std::bad_alloc)
    {
        // Failed to allocate
        if ( m_pLogger ) m_pLogger->ProgressFailure( LOG_PVS );
        return BCERR_OUTOFMEMORY;

    } // End if

    catch ( HRESULT& e )
    {
        // Arbitrary Error
        if ( m_pLogger && FAILED(e) ) m_pLogger->ProgressFailure( LOG_PVS );
        return e;

    } // End if

    // Success!!
    if ( m_pLogger ) m_pLogger->ProgressSuccess( LOG_PVS );
    return BC_OK;
}
```

Before we look at the RecursePVS function which will undoubtedly require heavy discussion, we will first examine the GetNextPortal method of this module that was used in the above code to select the next portal to have its PVS calculated.

GetNextPortal - CProcessPVS

This small function has the simple task of determining which portal in the list should have its PVS calculated next. The function loops through each portal in the list searching for the portal with the lowest PossbleVisCount that has not yet had its PVS calculated. That is, it is looking for the least complex portal in the list whose status is still PS_NOTPROCESSED. Once the portal is located, its status is set to PS_PROCESSING prior to its index being returned. This is the portal that will have its PVS calculated next and is therefore the portal that we are currently processing. The complete code is shown below.

```
ULONG CProcessPVS::GetNextPortal( )
{
    CPVSPortal * pPortal;
    long PortalIndex = -1, Min = 999999, i;

    // Loop through all portals
    for ( i = 0; i < (signed)GetPVSPortalCount(); i++ )
    {
        pPortal = GetPVSPortal(i);

             // If this portal's complexity is the lowest and it has
        // not already been processed then we could use it.
        if ( pPortal->PossibleVisCount < Min && pPortal->Status == PS_NOTPROCESSED)
        {
                    Min = pPortal->PossibleVisCount;
                    PortalIndex = i;
        } // End if Least Complex

     } // Next Portal

     // Set our status flag to currently being worked on =)
     if ( PortalIndex > -1) GetPVSPortal( PortalIndex )->Status = PS_PROCESSING;

    // Return the next portal
    return PortalIndex;
}
```

RecursePVS - CProcessPVS

We finally get to the function that contains the real meat of the process. This function calls itself recursively until the PVS set of the source portal has been calculated. The function takes three parameters. The first is the index of the leaf that we have currently stepped into which will be the neighbor leaf of the source portal the first time it is called. This leaf will be marked as visible in the source portals PVS. As we have stepped into this leaf it is obviously visible from the source portal. The second parameter is the source portal itself which is in the process of having its PVS calculated by this function. The third parameter is a PVSData structure which in the normal case will contain information about the target portal that is to be used in this function (the generator portal selected in the previous recursion) and the visibility buffer of all the portals we have stepped through so far which will be used for early-out determination.

```
HRESULT CProcessPVS::RecursePVS( ULONG Leaf, CPVSPortal * SourcePortal, PVSDATA & PrevData )
{
    ULONG           i,j;
    bool            More;
    ULONG          *Test, *Possible, *Vis;

    PVSDATA         Data;
    CPVSPortal     *GeneratorPortal;
    CPlane3         ReverseGenPlane, SourcePlane;
    CPortalPoints  *SourcePoints, *GeneratorPoints, *NewPoints;

    // Store the leaf for easy access
    CBSPLeaf * pLeaf = m_pTree->GetLeaf( Leaf );

    // Mark this leaf as visible
    SetPVSBit( SourcePortal->ActualVis, Leaf );
```

The first thing the function does is used the passed leaf index to fetch the leaf structure from the BSP tree. It then sets this leaf's bit to one in the source portal's ActualVis array. This leaf is visible from the source portal and has now been added to its PVS.

Notice in the above code that we instantiate on the stack a PVSData structure (in addition the one passed into the function as the third parameter). This new PVSData structure (imaginatively named Data) will be used to carry the generator portal information and combined visibility bitset from this function into the next recursion. That is, the Data local variable will be what we pass into the next call to the function as the third parameter and is the structure whose data we must fill out before we do so.

First, we allocate the structure a new VisBits array large enough to hold a bit for each leaf. This array will be used to store the result of combining the VisBits array passed into the function (PrevData.VisBits) with the PossibleVis array of the generator portal that we select in this function.

```
    // Allocate our current visibility buffer
    Data.VisBits = new UCHAR[ m_PVSBytesPerSet ];
    if (!Data.VisBits) throw std::bad_alloc(); // VC++ Compat
```

For ease of access, we also assign some local variables to structure members we are going to need to access frequently. We can see in the following code that the Possible local variable is used to point at the Data.VisBits array we just allocated and the Vis local variable is used to point at the PVS set (ActualVis array) of the source portal. This allows us to access these two buffers in short hand.

```
    // Store data we will be using inside the loop
    Possible    = (ULONG*)Data.VisBits;
    Vis         = (ULONG*)SourcePortal->ActualVis;
    GetPortalPlane( SourcePortal, SourcePlane );
```

Notice in the above code how we then fetch the plane of the source portal and store it in the SourcePlane local variable. Remember once again why the GetPortalPlane method is used for this task. It takes care of flipping the direction of the portal's node plane (the returned plane) if it faces into an opposing half space to the portals neighbor leaf.

The rest of the function is contained inside a loop that iterates through every portal in the current leaf and processes it for generator portal candidacy. Remember that the leaf structure in the BSP tree (which

we just retrieved above) will contain the indices of the two-way portals that reside in that leaf. As we did before, we must multiply this index by 2 so that we get the index of the first in the pair of one-way portals that were generated by that two-way portal. We then test to see if the first portal in the pair does not have a neighbor leaf equal to the leaf we are currently in because if it does, this is the one-way portal that flows into the current leaf and not out of it. When this is the case we know that the next portal in the array must be the other portal in the pair which faces out of the current leaf and into the neighbor leaf we may be interested in recurring into if this generator portal is not completely clipped away later in the function.

```
    // Check all portals for flow into other leaves
    for ( i = 0; i < pLeaf->PortalIndices.size(); i++ )
    {
        // Find correct portal index (the one IN this leaf (not Neighbouring))
        ULONG PortalIndex = pLeaf->PortalIndices[ i ] * 2;
        if ( GetPVSPortal( PortalIndex )->NeighbourLeaf == Leaf ) PortalIndex++;

        // Store the portal for easy access
        GeneratorPortal = GetPVSPortal( PortalIndex );
```

Now that we have the generator portal we can see how the PrevData.VisBits array that is passed into this function is used. This contains a leaf bitset that has been generated by ANDing all the PossibleVis arrays of the portals we have traveled through to get to this current leaf. Therefore, only leaves with their bits set to one in this array are leaves that are visible from the source portal when looking through all the portals we have stepped through to get to this leaf. Therefore, we test to see if the generator portal's neighbor leaf has its bit set to one in this bitset. If it does not, it means the leaf on the opposing side of this portal is not possibly visible from the source portal so we have no interest in processing this generator portal any further. As you can see, when this is the case we simply skip this generator portal and continue to the next iteration of the generator portal loop.

```
        // We can't possibly recurse through this portal if it's neighbour
        // leaf is set to invisible in the target portals PVS
        if ( !GetPVSBit( PrevData.VisBits, GeneratorPortal->NeighbourLeaf ) ) continue;
```

Now it is time to combine the PrevData.VisBits array with the visibility array of the generator portal so that we can store the resulting bit set in Data.VisBits (aliased via the 'Possible' pointer) and send that onto a future recursion. It is this section of code that performs the bitset accumulation we have discussed in each recursion. What we are going to do is take the visibility set we have been passed into this function (PrevData.VisBits) and further refine it by the generator portal's PossibleVis bits. But wait!!! If the generator portal has already had its PVS generated such that its status is equal to PS_PROCESSED, we can refine this bit set even further by using its ActualVis array instead of its PossibleVis array which will allow us to hopefully carve of a great many more visible leaves out of the equation. Therefore, in the next section of code we first test the status of the generator portal and if it has been processed, we assign the local Test pointer to point at its ActualVis array. If not, then Test is assigned to point at its PossibleVis array.

```
        // If the portal can't see anything we haven't already seen, skip it
        if ( GeneratorPortal->Status == PS_PROCESSED )
            Test = (ULONG*)GeneratorPortal->ActualVis;
        else
            Test = (ULONG*)GeneratorPortal->PossibleVis;
```

Now we will loop through PrevData.VisBits four bytes at a time (using a long pointer which is why we padded these visibility arrays to a 32 bit boundary) and will AND it with the visibility buffer of the generator portal 'Test'. We will store the result in 'Possible' which is a pointer to Data.VisBits which we know is the PVSData structure that will be passed into the next recur of this function. As we AND these arrays together four bytes at a time and store them in 'Possible', we will also perform a bitwise AND with the NOT of the source portals PossibleVis array (Aliased by the local Vis pointer). This allows us to quickly detect if there are any bits set in Possible which are also set in Vis and therefore means there are still leaves in Possible that might be visible from the source portal and will need to be visited. When this is the case we set the More local Boolean variable to true

```
        More = false;
        // Check to see if we have processed as much as we need to
        // this is an early out system. We check in 32 bit chunks to
        // help speed the process up a little.
        for ( j = 0; j < m_PVSBytesPerSet / sizeof(ULONG); j++ )
        {
            Possible[j] = ((ULONG*)PrevData.VisBits)[j] & Test[j];
            if ( Possible[j] & ~Vis[j] ) More = true;

        } // Next 32 bit Chunk

        // Can we see anything new ??
        if ( !More ) continue;
```

The little loop above can be a bit of a brain teaser at first but it is essentially just creating the combined VisBits array that will be sent into the next recursion and at the same time determining whether there is anything left to see by continuing down this path. Outside the loop you can see that if More is not set to true, if means there are no bits set to one in the Possible array that are also set to one in the source portal's PossibleVis array and as such, there is nothing of interested on the other side of this generator portal so there is not need to recur through it into the neighbor leaf. When this is the case we simply skip any further processing of the generator portal and continue to the next iteration of the loop.

In the next step we fetch the plane of the generator portal and store it in the Data.TargetPlane so that it can be passed to the next recursion as the target portal's clip plane. Remember, the generator portal selected in this recursion will become the next target portal when we recur through it into its neighbor leaf.

```
        // The current generator plane will become the next recursions target plane
        GetPortalPlane( GeneratorPortal, Data.TargetPlane );
```

Our next test is to make sure that the generator portal is not ON_PLANE with the target portal that has been passed into this function in the PrevData parameter. If it is, then they cannot see each other and we can skip this generator portal. For this test we are using a generator plane with a reversed normal and are comparing this normal against the normal the normal of the target portal (PrevData.TargetPlane). We do

this because we need to check that the portal we have just entered this leaf through is not on the same plane as any we are about to leave this leaf through. A target portal can not see a generator portal that is on the same plane as it and stepping through such a portal could carry us back into the leaf that we just exited in the previous recur.

```
        // We can't recurse out of a coplanar face, so check it
        ReverseGenPlane.Normal   = -Data.TargetPlane.Normal;
        ReverseGenPlane.Distance = -Data.TargetPlane.Distance;
        if ( ReverseGenPlane.Normal.FuzzyCompare( PrevData.TargetPlane.Normal, 0.001f ))
              continue;
```

Our next task is to clip the generator portal to the plane of the source portal so that any portion of the generator portal that is contained in the back space of the source portal is removed. If this operation completely clips away the generator portal we skip this generator portal as there seems to be no valid portion of it in front of the source portal.

```
        // Clip the generator portal to the source. If none remains, continue.
        GeneratorPoints = GeneratorPortal->Points->Clip( SourcePlane, false );
        if ( GeneratorPoints != GeneratorPortal->Points )
             FreePortalPoints( GeneratorPortal->Points );

        if (!GeneratorPoints) continue;
```

In this next section of code we see some special case code which is only executed the first time the RecursePVS function is called (when PrevData.TargetPoints will equal NULL). When this is the case we do not wish to perform any anti-penumbra clipping as we are really just trying to select the first target portal. As you can see, in this instance we simply copy over the source points passed into the function (PrevData.SourcePoint) into the PVSData structure that will be passed into the next recursion (Data.SourcePoints) and also copy over the generator portal into the TargetPoints member of this structure also so that it will become the target portal in the next recursion. We then call the RecursePVS function to recur into the neighbor leaf of the generator portal (which is really the first target portal in this instance) and on function return skip to the next generator portal in the leaf to be processed.

```
        // The second leaf can only be blocked if coplanar
        if ( !PrevData.TargetPoints )
        {
            Data.SourcePoints = PrevData.SourcePoints;
            Data.TargetPoints = GeneratorPoints;
            RecursePVS( GeneratorPortal->NeighbourLeaf, SourcePortal, Data );
            FreePortalPoints( GeneratorPoints );
            continue;

        } // End if Previous Points
```

Remember that the above conditional code is only executed in the first instance of the function when no previous target portals exist and need to be found.

The next thing we do is clip the generator portal to the target portal's plane. The section of the generator portal that should be visible to the source portal is that section located in the front space of the target portal and as such, can be seen when looking through the back of the target portal. If none of the generator portal survives the clip then it is not visible to the source portal and we do not have to step

through it into its neighbor leaf. When this is the case we process the generator portal no further and simply skip to the next generator portal in the loop that needs to be processed.

```
        // Clip the generator portal to the previous target. If none remains, continue.
        NewPoints = GeneratorPoints->Clip( PrevData.TargetPlane, false );
        if ( NewPoints != GeneratorPoints ) FreePortalPoints( GeneratorPoints );
        GeneratorPoints = NewPoints;
        if (!GeneratorPoints) continue;
```

The source portal should also be clipped to the generator portals plane. However, we need to make sure that the only portion of the source portal that survives is the portion that is located behind the plane of the generator portal and not in front as it usually the case with a clipping routine. The source portal should only ever be able to see through the back of a generator portal and a source portal that spans the generator portals plane clearly violates that. We need to flip our clipping operation so that is clips away any portion of the source portal that is located in the front space of the generator portal's plane. Fortunately, we already calculated a reversed generator plane above so we can re-use it here. By passing in the reversed generator plane into our clip function we will remove the section of the source portal that is in the back space of this plane, which is really the section that is in the front space of the real generator portal's plane. As you can see, we make a new copy of the source portal geometry from the source portal geometry passed into the function (PrevData.SourcePoint) which my have been clipped many times before and we then clip it to the reversed generator plane.

```
        // Make a copy of the source portals points
        SourcePoints = new CPortalPoints( PrevData.SourcePoints, true );

        // Clip the source portal
        NewPoints = SourcePoints->Clip( ReverseGenPlane, false );
        if ( NewPoints != SourcePoints ) FreePortalPoints( SourcePoints );
        SourcePoints = NewPoints;

        // If none remains, continue to the next portal
        if ( !SourcePoints ) { FreePortalPoints(  GeneratorPoints ); continue; }
```

As the above code shows, if none of the source portal survived the clip it means the source portal can see no portion of the generator portal's back face and as such no portal flow can exist. When this is the case we skip any further processing of this generator portal and continue to the next iteration of the loop where we will test the next generator portal in this leaf which needs to be tested.

At this point we know we have a source portal and a generator portal which although have been clipped to each others plane, have portal flow between them in the loosest sense. Our next task is to perform the clipping of the source and generator portals to the anti-penumbra. The ClipToAntiPenumbra function is the method that creates the anti-penumbra and performs the clipping of the passed portal to its planes. As discussed earlier, there are a possibly four anti-penumbras that we can build and clip to and how many we perform depends on the number of clip tests that have been enabled in the PVSOPTIONS structure. The clip tests are performed as shown below:-

- Clip Test 1 : Build Anti-Penumbra from **source portal** to **target portal** and clip the **generator portal** to its planes.
- Clip Test 2 : Build Anti-Penumbra from **target portal** to **source portal** and clip the **generator portal** to its planes.
- Clip Test 3 : Build Anti-Penumbra from **generator portal** to **target portal** and clip the **source portal** to its planes.
- Clip Test 4 : Build Anti-Penumbra from **target portal** to **generator portal** and clip the source portal to its planes.

After each clipping operation, if none of the portal that was clipped survived, the source and generator portals can not see each other through the target portal meaning we need process this generator portal no further and should not recur through it into the neighbor leaf. Instead, we simply continue on to the next iteration of the loop and process the next generator portal in the leaf. Here is the code that performs the four clip tests. We will look at the ClipToAntiPenumbra method next.

```
        // Lets go Clipping :)
        if ( m_OptionSet.ClipTestCount > 0 )
        {
            GeneratorPoints = ClipToAntiPenumbra( SourcePoints,
                                                  PrevData.TargetPoints,
                                                  GeneratorPoints,
                                                  false );

            if (!GeneratorPoints) { FreePortalPoints( SourcePoints ); continue; }

        } // End if 1 Clip Test

        if ( m_OptionSet.ClipTestCount > 1 )
        {
            GeneratorPoints = ClipToAntiPenumbra( PrevData.TargetPoints,
                                                  SourcePoints,
                                                  GeneratorPoints,
                                                  true );

            if (!GeneratorPoints) { FreePortalPoints( SourcePoints ); continue; }

        } // End if 2 Clip Tests

        if ( m_OptionSet.ClipTestCount > 2 )
        {
            SourcePoints = ClipToAntiPenumbra( GeneratorPoints,
                                               PrevData.TargetPoints,
                                               SourcePoints,
                                               false );

            if (!SourcePoints) { FreePortalPoints( GeneratorPoints ); continue; }

        } // End if 3 Clip Test

        if ( m_OptionSet.ClipTestCount > 3 )
        {
            SourcePoints = ClipToAntiPenumbra( PrevData.TargetPoints,
```

```
                                            GeneratorPoints,
                                            SourcePoints,
                                            true );

            if (!SourcePoints) { FreePortalPoints( GeneratorPoints ); continue; }

        } // End if 4 Clip Test
```

The first two parameters to the ClipToAntiPenumbra function are the portals between which the anti-penumbra will be created. The third parameter is where the portal that should be clipped is passed.

The fourth parameter to the function is a Boolean that describes whether the portal should be clipped to the back of the anti-penumbra planes or the front. This is necessary because the direction of the plane normals being built from source to target will be opposite to that of the normals of the plane constructed when going from target to source as is shown in figure 17.6.

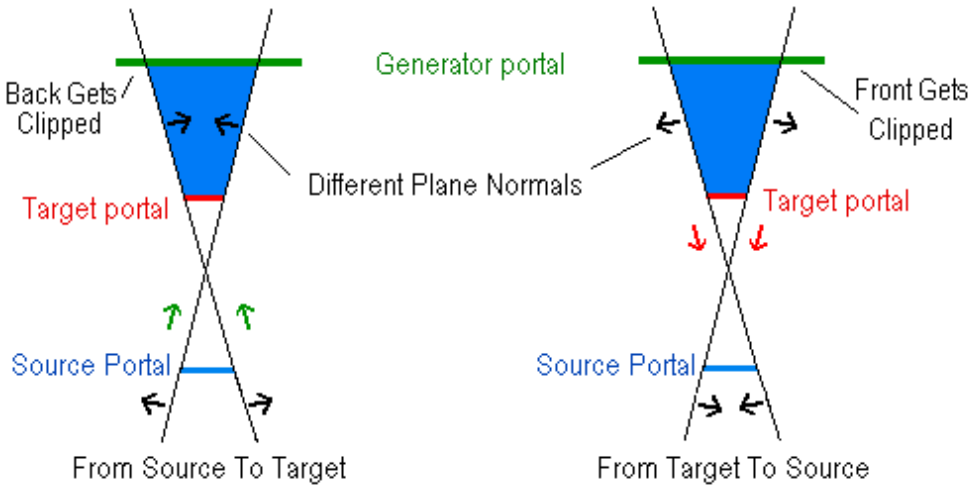


Figure 17.55

The leftmost image shows a 2D representation of the planes generated when the anti-penumbra is being constructed from the vertices of the source portal to the edges of the target portal. We can see that on the opposite side of the target portal the visible region is bounded by planes that face inwards and as such, any portion of the portal that is to be clipped that is located in the back space of any anti-penumbra plane is clipped away. In the right most image we see that when the direction of the anti-penumbra is reversed and is instead constructed from the target portal to the source portal, the visible region on the opposite side of the target portal is now bounded by outward facing planes and as such, the function should clip away any section of the generator portal that is located in the front space of any of these planes. That is why we flick this Boolean switch with each call in the above section of code. We are informing the ClipToAntiPenumbra method as to whether the portal should be clipped to the back or front spaces of the anti-penumbra's clip planes respectively.

If we survive the clip tests performed in the previous section of code then it means the source portal can clearly see the generator portal and as such, we should recur through the generator portal into its neighbor leaf. Before doing that we store the new clipped source portal polygon and the clipped generator portal polygon in the PVSData structure (Data) and then pass it into the RecursePVS function to recur into the next leaf.

Below we show the remainder of the function that performs this task and shows the clean up of the temporarily clipped source and generator portals inside the bottom of the loop.

```
        // Store data for next recursion
        Data.SourcePoints = SourcePoints;
        Data.TargetPoints = GeneratorPoints;

        // Flow through it for real
        RecursePVS( GeneratorPortal->NeighbourLeaf, SourcePortal, Data );

        // Clean up
        FreePortalPoints( SourcePoints );
        FreePortalPoints( GeneratorPoints );

    } // Next Portal

    // Clean up
    if (Data.VisBits) delete []Data.VisBits;

    // Success
    return BC_OK;
}
```

When we reach the bottom of this loop we will have processed every generator portal in this leaf and will have calculated its contribution to the source portal's ActuaVis array. Before returning we release the Data.VisBits array which we allocated earlier.

ClipToAntiPenumbra - CProcessPVS

The ClipToAntiPenumbra function has the task of finding all separating planes that divide the source and target portals into opposing half spaces. Such a plane is a valid anti-penumbra plane which is used to clip the generator portal.

> **Note:** In this function we are using the term generator portal to refer to the portal passed in as the third parameter to this function. However, as we saw in the above code, this may be a pointer to either the actual source or generator portal depending on which clip test is being performed.

This function first sets up a loop to iterate through every edge in the source portal as shown below.

```
CPortalPoints * CProcessPVS::ClipToAntiPenumbra( CPortalPoints * Source,
                                                 CPortalPoints * Target,
                                                 CPortalPoints * Generator,
                                                 bool ReverseClip )
{
    CPlane3         Plane;
    CVector3        v1, v2;
    float           Length;
    ULONG           Counts[3];
    ULONG           i, j, k, l;
    bool            ReverseTest;
    CPortalPoints  *NewPoints;

    // Check all combinations
    for ( i = 0; i < Source->VertexCount; i++ )
    {
```

```
        // Build first edge
        l = ( i + 1 ) % Source->VertexCount;
        v1 = Source->Vertices[l] - Source->Vertices[i];
```

Loop variable ‘i’ describes the index of the first vertex in the edge we are testing and the ‘l’ local variable describes the index of the next vertex forming the edge. This is calculated as being i+1 with a modulus performed with the vertex count of the source polygon so that ‘l’ will wrap around to the first vertex for the last edge of the portal. Vector v1 is then calculated by subtracting vertex position l from i thus generating the current edge vector in the source portal we wish to process.

Now that we have an edge in the source portal we need to create a plane with that edge and every vertex in the target portal. Next we set up a loop to loop through each vertex in the target portal and create an additional vector v2. This is a vector from the one of the source vertices in the edge and the current vertex in the target portal we are processing. Vectors v1 and v2 are now vectors tangent to a plane generated from the source to the target portal so we will next perform the cross product between these two vectors to retrieve the normal of that plane.

```
        // Find a vertex belonging to the generator that makes a plane
        // which puts all of the vertices of the target on the front side
        // and all of the vertices of the source on the back side
        for ( j = 0; j < Target->VertexCount; j++ )
        {
            // Build second edge
            v2 = Target->Vertices[ j ] - Source->Vertices[ i ];
            Plane.Normal = v1.Cross( v2 );
```

Next we record the length of the returned normal (which has not yet been normalized) and if the length is found to be zero (with tolerance) it means we have an invalid plane so will continue to the next iteration of the loop where we will process the next target vertex.

```
            // If points don't make a valid plane, skip it
            Length = Plane.Normal.x * Plane.Normal.x +
                     Plane.Normal.y * Plane.Normal.y +
                     Plane.Normal.z * Plane.Normal.z;
            if ( Length < 0.1f ) continue;
```

If we get this far it means that we have a valid plane and as such we will normalize the plane normal and will calculated the plane’s distance from the origin by dotting the plane normal with the target vertex (which is a point known to be on the plane).

```
            // Normalize the plane normal
            Length = 1 / sqrtf( Length );
            Plane.Normal *= Length;

            // Calculate the plane distance
            Plane.Distance = -Target->Vertices[ j ].Dot( Plane.Normal );
```

At this point we have a valid plane so our next step is to test if this is a separating plane. That is, if it has the source portal completely contained in one half space and the target portal in the other. If not then this is not a valid an anti-penumbra plane. Figure 17.7 reminds us of what a separating plane looks like by

showing us both a separating plane and a non-separating plane. Only the plane generated in the leftmost image is a valid anti-penumbra clip plane and will be used to clip the generator portal.

Figure 17.7

Our first task in achieving this goal is to loop through each vertex in the source portal and classify it against our plane. As it was one of the source portal's vertices that was used to generate this plane, it is impossible to have vertices in the source portal located in both half spaces of the plane. Therefore, in this loop we are searching for the first vertex that is found in either the front or back half spaces at which point we can break. We know that if one of the vertices is contained in the front space of the plane then the entire polygon must be and likewise for the back half space. If the source portal is found to be located in the back half space of the plane we set the ReverseTest Boolean to false before breaking so that we know that when we test the target portal we are wishing it to be contained in the front half space. Alternatively, this Boolean is set to true if the source portal is found to be located in the front space of the plane meaning we wish to find the target portal located in the back space.

```
            // Find out which side of the generated separating plane has the source portal
            ReverseTest = false;
            for ( k = 0; k < Source->VertexCount; k++ )
            {
                // Skip if it matches other verts
                if ( k == i || k == l ) continue;

                // Classify the point
                CLASSIFYTYPE Location = Plane.ClassifyPoint( Source->Vertices[ k ] );
                if ( Location == CLASSIFY_BEHIND )
                {
                    // Source is on the negative side, so we want all pass
                    // and target on the positive side.
                    ReverseTest = false;
                    break;

                } // End If Behind
                else if ( Location == CLASSIFY_INFRONT )
                {
                    // Source is on the positive sode, so we want all pass
                    // and target on the negative side.
                    ReverseTest = true;
                    break;

                } // End if In Front

            } // Next Source Vertex
```

As the above loop has no conditional to deal with the on-plane case, if all its vertices are co-planar with the candidate plane then the loop will be allowed to carry out to its conclusion. This means outside the loop, the loop variable ‘k’ will be equal to the vertex count of the source portal. This tells us that the source portal is co-planar and as such, this plane could not possibly be a separating plane so we should skip it and continue on to test the next vertex in the target portal.

```
            // Planar with the source portal ?
            if ( k == Source->VertexCount ) continue;
```

In the next section we will loop through each vertex in the target portal and will classify it against the plane. Before doing so we test the value of the ReverseTest local Boolean and flip the direction of the plane normal if it is set to true. This allows us to treat the source portal as having existed in the back space of the plane (regardless of where it was actually located) which means in this code we are searching for a portal that has its vertices contained in the front space of the plane.

```
            // Flip the normal if the source portal is backwards
            if ( ReverseTest ) { Plane.Normal = -Plane.Normal;
                                 Plane.Distance = -Plane.Distance; }

            // If all of the pass portal points are now on the positive
            // side then this is the separating plane.
            ZeroMemory( Counts, 3 * sizeof(ULONG) );
            for ( k = 0; k < Target->VertexCount; k++ )
            {
                // Skip if the two match
                if ( k == j ) continue;

                // Classify the point
                CLASSIFYTYPE Location = Plane.ClassifyPoint( Target->Vertices[ k ] );
                if ( Location == CLASSIFY_BEHIND )
                    break;
                else if ( Location == CLASSIFY_INFRONT )
                    Counts[0]++;
                else
                    Counts[2]++;

            } // Next Target Vertex

            // Points on the negative side ?
            if ( k != Target->VertexCount ) continue;

            // Planar with separating plane ?
            if ( Counts[0] == 0 ) continue;
```

As you can see, as soon as we find a vertex that is situated behind the plane it means this portal is in the same half space of the plane as the source portal (pay attention to the flipping of the plane depending on the result of ReverseTest that allows us to make that determination). This means we break instantly as this is not a separating plane. If any vertices are found to exist in the front space of the plane then we increase the value of Counts[0].

Outside the loop we can see that if loop variable ‘k’ is not equal to the vertex count of the target portal it means we broke from the loop early after finding a vertex in the same half space of the source portal. This means this plane can not possibly separate the source and target portals into two half spaces so we continue on to the next iteration of the loop and the next plane to test. We can also see that if Counts[0]

equal zero, it means we never incremented it in the loop meaning no vertex was every found to be in the front space of the plane either. This must mean the target portal is located on the candidate plane and as such, this can not possibly be a separating plane. Once again, if this is the case we skip any further processing of this plane and skip to the next iteration of the loop.

If we reach this point it means we have found a separating plane and the generator portal passed into the function should be clipped to it. Before doing so we test the value of the ReverseClip parameter remembering that if this is set to true then the caller would like us to clip away portal fragments that are in the front space of the separating plane instead of in the back space as is usually the case. We address this by simply flipping the direction of the clip plane prior to clipping the portal.

As the next and final section of this function shows, we clip the generator portal to the clip plane and if we find that a new portal was created by the clip procedure, the original portal (Generator) is released. We then assign the passed Generator portal pointer to point at the new clipped polygon returned from the clipping function instead. In the last line of the loop for this plane, we test that the Generator has not been assigned a value of NULL and if so, it means we must have completely clipped away the generator portal with this plane and as such, the generator portal is outside the anti-penumbra and NULL is returned.

```
            // Flip the normal if we want the back side
            if ( ReverseClip ) { Plane.Normal = -Plane.Normal;
                                 Plane.Distance = -Plane.Distance; }

            // Clip the target by the separating plane
            NewPoints = Generator->Clip( Plane, false );
            if ( NewPoints != Generator ) FreePortalPoints( Generator );
            Generator = NewPoints;

            // Target is not visible ?
            if (!Generator) return NULL;

        } // Next Target Vertex

    } // Next Source Vertex

    // Success!!
    return Generator;
}
```

The bottom of the function is only ever reached if some portion of the generator portal survived being clipped to all the separating planes generated in the above loops. This new clipped generator portal is returned from the function back to the caller.

Where Are We?

At this point we have covered the core PVS generation procedure that was invoked from the CProcessPVS::Process method via a call to the CalcPortalVis method. When the CalcPortalVis method returns program flow back to the Process method, every portal will have had its PVS generated. Let us have another look at the CProcessPVS::Process method to remind us of what is let to do.

```
HRESULT CProcessPVS::Process( CBSPTree * pTree )
{
    HRESULT hRet;

    // Validate values
    if (!pTree) return BCERR_INVALIDPARAMS;

    // Validate Input Data
    if ( pTree->GetPortalCount() == 0 ) return BCERR_BSP_INVALIDTREEDATA;

    // Store tree for compilation
    m_pTree = pTree;

    // Calculate Number Of Bytes needed to store each leafs
        // vis array in BIT form (i.e 8 leafs vis per byte uncompressed)
    m_PVSBytesPerSet = (pTree->GetLeafCount() + 7) / 8;

    // 32 bit align the bytes per set to allow for our early out long conversion
    m_PVSBytesPerSet = (m_PVSBytesPerSet * 3 + 3) & 0xFFFFFFFC;

    // Retrieve all of our one way portals
    hRet = GeneratePVSPortals();

    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;

    // Calculate initial portal visibility
    hRet = InitialPortalVis();

    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;

    // Perform actual full PVS calculation
    hRet = CalcPortalVis();

    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;

    // Export the visibility set to the final BSP Tree master array
    hRet = ExportPVS( pTree );

    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;

    // Success
    return BC_OK;
}
```

As you can see, we have covered all methods in the process with the exception of the final one called ExportPVS. It is this method, which we will look at next, that is responsible for combining the PVS data

for each portal that resides in a given leaf to generate the PVS for that leaf in the BSP tree. This method will also compress the data using zero run length encoding and will store in the leaves of the BSP tree the index into which each leaf's visibility data begins in the master PVS data block

ExportPVS - CProcessPVS

At the time this function is called each portal will contain its own PVS in its ActualVis array. However, our PVS rendering code will not be interested in what each of the portals can see and in fact, there is no need for the portals to even be saved to file. Our PVS renderer wants to know what each leaf can see and it is this function that takes care of building the master PVS data block that will be stored in the BSP tree and will describe the visibility sets for each leaf instead of each portal.

Now that we have the visibility sets for each portal, determining what each leaf can see is trivial. The PVS of a leaf is simply the accumulation of the PVS's of each portal that resides in that leaf and as such can be calculated with a simple loop at each leaf.

Let us cover the function a section at a time. The first section of the function writes out logging information to the PVS channel describing whether it is going to create a compressed leaf PVS or whether it will calculate the leaf PVS for the BSP tree as an uncompressed bit set (this is controller by a #define).

```
HRESULT CProcessPVS::ExportPVS( CBSPTree * pTree )
{
    UCHAR * PVSData = NULL;
    UCHAR * LeafPVS = NULL;
    ULONG   PVSWritePtr = 0, i, p, j;

    try
    {
        // ************************
        // * Write Log Information *
        // ************************
        if ( m_pLogger )
        {

            #if ( PVS_COMPRESSDATA )
                m_pLogger->LogWrite( LOG_PVS,
                                     0,
                                     true,
                                     _T("ZRLE compressing PVS data for export \t\t- " ) );
            #else
                m_pLogger->LogWrite( LOG_PVS,
                                     0,
                                     true,
                                     _T("Building final PVS data for export \t\t- " ) );
            #endif

            m_pLogger->SetRewindMarker( LOG_PVS );
            m_pLogger->LogWrite( LOG_PVS, 0, false, _T("0%%" ) );
            m_pLogger->SetProgressRange( pTree->GetLeafCount() );
            m_pLogger->SetProgressValue( 0 );
        }
        // ************************
        // *    End of Logging    *
```

```
        // ************************
```

Notice that the progress range of the logger in the above code is set to the number of leaves in the passed BSP tree. That is because this process will be complete once we have looped through each leaf in the BSP tree and have calculated its PVS and added it to a master PVS array.

Our next step is to allocate some memory that will be used to store the master PVS data array. This must be large enough to store the visibility for every leaf in the tree in compressed format. One might imagine that if m_PVSBytesPerSet contains the number of bytes needed to store a leaf's visibility information in uncompressed format, then we can allocate an array large enough to store LeafCount*m_PVSBytesPerSet bytes. This should always be large enough to store everything we need as it is large enough to store a visibility set for each leaf in uncompressed format. Is that correct?

Well, in practice yes that will almost definitely be the case and in fact this array will be much larger than we actually need once the data is compressed. That doesn't matter though because after we have compressed the data we can resize the array to its correct actual size. However, look at how we allocate this array and initialize its memory.

```
        // Reserve Enough Space to hold every leafs PVS set
        PVSData = new UCHAR[pTree->GetLeafCount() * (m_PVSBytesPerSet*2)];
        if (!PVSData) throw std::bad_alloc();

        // Set all visibility initially to off
        ZeroMemory( PVSData, pTree->GetLeafCount() * (m_PVSBytesPerSet*2));
```

Why are we multiplying m_PVSBytesPerSet by 2 before multiplying it with the tree's leaf count? Although incredibly unlikely, there is very slim chance that compressing our data using ZRLE could actually make it larger than in its uncompressed format. Imagine for example we had 6 bytes in each of our visibility sets and the 2^{nd}, 4^{th} and 6^{th} bit were set to zero.

Leaf Set = N , 0 , N , 0 , N , 0

Imagining that N is some placeholder for a non zero byte in a leaf's PVS. We know that ZRLE encoding will try to compress runs of zero bytes by collapsing those runs in to two bytes. The first is the zero itself and the second is the byte that tells us how many bytes of zero the run represents. However, in the above scenario there are no runs of zeros and as such, an additional byte (the run byte) would be inserted after each zero describing a run of 1. Obviously this provides no benefit and actually would increase the size of our 6 byte visibility array to a 9 byte array.

Leaf Set = N , 0 , 1 , N , 0 , 1 , N , 0 , 1

Therefore we can see that in the most unusual circumstances we may generate a compressed PVS that is larger than its uncompressed counterpart and in the most appalling of circumstances, compressing the PVS could actually double its size.

> **Note:** it should be noted that the above scenario is very unlikely as the PVS by its very nature in an occluded environment will generate leaf based PVS sets where most of the other leaves of not visible and as such, huge runs of zeros will be compressed into only two bytes providing us with huge memory

savings in nearly every case. However, as it is ***possible*** that we could double its size by compressing it, we had better allocate the PVS data block large enough to handle it so that our compiler does not crash and burn mid way through the compression procedure.

With the master PVS data array allocated we will now allocate a temporary buffer that will be used by each leaf to accumulate and collect the combined ActualVis arrays of each portal contained in that leaf. This buffer should be large enough to store an uncompressed bit set for each leaf. That is, this buffer should be equal in size to the portal's ActualVis array.

```
        // Allocate enough memory for a single leaf set
        LeafPVS = new UCHAR[ m_PVSBytesPerSet ];
        if (!LeafPVS) throw std::bad_alloc();
```

Now we will loop through each leaf in the tree. Inside the loop we will use the loop variable to fetch from the BSP tree the structure of the leaf we are currently processing. We also update the progress of the logger which is incremented on a per leaf bases.

```
        // Loop round each leaf and collect the vis info
        // this is all OR'd together and ZRLE compressed
        // Then finally stored in the master array
        for ( i = 0; i < pTree->GetLeafCount(); i++ )
        {
            CBSPLeaf * pLeaf = pTree->GetLeaf(i);

            // Update progress
            if (!m_pParent->TestCompilerState()) break;
            if ( m_pLogger ) m_pLogger->UpdateProgress( );
```

In the next step we will initialize the temporary LeafPVS buffer to zero prior to collecting the visibility sets of each of its portals into it. We also set the PVSIndex member of the leaf to that of the value stored in the PVSWritePtr local variable. We will see in a moment that this variable will be increased each time we compress the PVS data for a leaf and add it to the master PVS data block and as such, it will always describe the index into this array where the compressed PVS data of the leaf we are about to process will be placed into this array. We also set the visibility bit in the LeafPVS buffer for the current leaf we are processing as it can obviously see itself.

Note: PVSWritePtr is zero the first time this loop executes which describes leaf 0's PVS as starting at the very beginning of the master PVS data array

```
            // Clear Temp PVS Array Buffer
            ZeroMemory( LeafPVS, m_PVSBytesPerSet );
            pLeaf->PVSIndex = PVSWritePtr;

            // Current leaf is always visible
            SetPVSBit( LeafPVS, i );
```

Now we loop through every portal in this leaf and use the familiar methods to retrieve the index of the one-way portal that it represents.

```
            // Loop through all portals in this leaf
            for ( p = 0; p < pLeaf->PortalIndices.size(); p++ )
            {
```

```
                // Find correct portal index (the one IN this leaf)
                ULONG PortalIndex = pLeaf->PortalIndices[ p ] * 2;
                if ( GetPVSPortal( PortalIndex )->NeighbourLeaf == i ) PortalIndex++;
```

At this point we have a pointer to the current portal so we will loop through each byte in its ActualVis array and will bitwise OR it with the current contents of the LeafPVS buffer so that the necessary visibility bits are enabled.

```
                // Or the vis bits together
                for ( j = 0; j < m_PVSBytesPerSet; j++ )
                {
                        LeafPVS[j] |= GetPVSPortal( PortalIndex )->ActualVis[j];

                } // Next PVS Byte

            } // Next Portal
```

At this point the LeafPVS buffer contains the complete uncompressed PVS set for the current leaf that we are processing so our next task is to store this in the master PVSData array. If compression has not been enabled then we literally just copy the information stored in LeafPVS into the master data array at the location contained in PVSWritePtr and then increase the write pointer by the size of the LeafPVS array. This is done so that when we process the next leaf, it will contain an index to the location where its data should be copied to in the master PVS data array. However, if we have chosen to compress the data set then the data is added to the master PVSData array using a function called CompressLeafSet as shown below.

```
            #if ( PVS_COMPRESSDATA )

                // Compress the leaf set here and update our master write pointer
                PVSWritePtr += CompressLeafSet( PVSData, LeafPVS, PVSWritePtr );

            #else

                // Copy the data into the Master PVS Set
                memcpy( &PVSData[ PVSWritePtr ], LeafPVS, m_PVSBytesPerSet );
                PVSWritePtr += m_PVSBytesPerSet;

            #endif

        } // Next Leaf
```

The CompressLeafSet method will be discussed in a moment but for now just know that it is passed the master PVSData array as its first parameter as this is where the function will need to copy the compressed data into. As the second parameter we will pass the LeafPVS buffer which contains the uncompressed PVS for the current leaf. This is the data that the function will compress and copy into the PVSData array. As the third parameter we pass the write pointer index value which describes the starting location in the PVSData array where the new compressed data should be written to. As this function returns the size that the passed LeafPVS data array was ultimately compressed to, we can use this value to increment the write pointer value on function return so that it contains the index of the first byte after the block of data we have just added. This will be the starting location for the next leaf's compressed data and the byte at which its PVS data will begin in the master array.

At this point we will have compressed every leaf's PVS and stored it in the PVSData array and each leaf in the tree will also have had its PVSIndex member set so that it describes the index of the first byte in this array where its visibility information begins. We can now delete the LeafPVS buffer as it is no longer needed and can call the BSP tree's SetPVSData method (discussed earlier) so that it can make a copy of the PVSData array we haven just compiled.

```
        // Clean up after ourselves
        delete []LeafPVS;
        LeafPVS = NULL;

        // Pass this data off to the BSP Tree (data, size, compressed)
        if (FAILED(pTree->SetPVSData( PVSData,
                                      PVSWritePtr,
                                      PVS_COMPRESSDATA ))) throw std::bad_alloc();

        // Free our PVS buffer
        delete []PVSData;

        // If we're cancelled, bail
        if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;

    } // End Try Block

    catch (...)
    {
        // Clean up and return (Failure)
        if ( LeafPVS ) delete []LeafPVS;
        if ( PVSData ) delete []PVSData;
        if ( m_pLogger ) m_pLogger->ProgressFailure( LOG_PVS );
        return BCERR_OUTOFMEMORY;

    } // End Catch Block

    // Success!!
    if ( m_pLogger ) m_pLogger->ProgressSuccess( LOG_PVS );
    return BC_OK;
}
```

Recall that the CBSPTree::SetPVSData method will make a copy of the passed PVS data array and by passing in PVSWritePtr as the second parameter we also inform it of the final size of the compressed data. As it uses this to allocate its PVS array this means that whilst the PVSData array allocated in this function may have been allocated much larger than necessary, the actual array stored in the PVS tree will be the correct size. The CBSPTree::SetPVSData method copies over all the PVS data into its own array and therefore, on function return we can delete the local PVSData array as it is no longer needed. The third parameter to the SetPVSData method simply informs the function that the data is compressed which the run time component will need to know when reading the PVS data at render time.

CompressLeafSet - CProcessPVS

This method is passed as its first parameter a master PVS data buffer that the compressed data will be copied into and as its 3rd parameter the location within this array where we should start writing the compressed data. As the second parameter the uncompressed PVS of a single leaf of passed. This is the data that is to be compressed.

The function first sets up a loop to iterate through every byte in the passed PVS set (VisArray).

```
ULONG CProcessPVS::CompressLeafSet ( UCHAR MasterPVS[],
                                     const UCHAR VisArray[],
                                     ULONG WritePos)
{
    ULONG   RepeatCount;
    UCHAR  *pDest = &MasterPVS[ WritePos ];
    UCHAR  *pDest_p;

    // Set dynamic pointer to start position
    pDest_p = pDest;

    // Loop through and compress the set
    for ( ULONG j = 0; j < m_PVSBytesPerSet; j++ )
    {
        // Store the current 8 leaves
        *pDest_p++ = VisArray[j];

        // Don't compress if all bits are not zero
        if ( VisArray[j] ) continue;
```

pDest_p is used to point at the current byte in the master PVS data array (passed as the first parameter) that we are currently copying information into. As you can see in the first line inside the loop in the above code, we copy the contents of the current byte being processed in the leaf set (VisArray) into the master PVS data array. In the bottom line in the above code we can see that after copying over this byte we test to see if it was zero or not. If it isn't zero then this byte has visible leaves in its bitset and can not be compressed. This means we can just skip to the next byte in the buffer having copied over this byte into the master array.

If we make it past the last line in the above code however, it means the current byte we have just copied over is a zero byte and therefore we must set up a loop to see how many zero bytes follow it. Once we have counted the run of zeros, we can simply insert this run length byte into the master PVS data array just after the zero byte we just copied over. Here is the remainder of the function.

```
        // Count the number of 0 bytes
        RepeatCount = 1;
        for ( j++; j < m_PVSBytesPerSet; j++ )
        {
            // Keep counting until byte != 0 or we reach our max repeat count
            if ( VisArray[j] || RepeatCount == 255) break; else RepeatCount++;

        } // Next Byte

        // Store our repeat count
        *pDest_p++ = (UCHAR)RepeatCount;
```

```
        // Step back one byte because the outer loop
        // will increment. We are already at the correct pos.
             j--;

    } // Next Byte

    // Return written size
    return pDest_p - pDest;
}
```

As you can see, we loop from our current byte position through the bytes that follow counting how many zero bytes we encounter in a continuous run. The RepeatCount is set to 1 to begin with as before this loop we only know of one zero that exists, the zero byte we just copied over. In the middle of this inner loop you can see that we break from the loop as soon as a non-zero byte is encountered or if the repeat count reaches 255 which is the highest run length value we can store in a byte. Otherwise, for each consecutive zero byte we find in the run we increment the repeat count.

Outside the inner loop, RepeatCount will contain the length of the run of zeros that were found starting from the original zero byte we copied into the master array. We then write this run length into the PVS data buffer (pDest_p).

Finally, we return from the function the total size of the data we managed to compress by subtracting from pDest_p, which contains the address of the last byte we have just written to the PVS data array, the value of pDest which contains the address of the first byte that was written by this function. That is, we return the number of bytes of compressed data that we have written to the PVS data array.

Conclusion

Writing a portal and PVS compiler has perhaps been our most challenging task to date. However, our compiler tool has now evolved into something quite significant. You are certainly not expected to grasp every nuance of the code we have written in this chapter after a single read through the work book however, using this book as an aid to help you navigate the source code will have you up to speed on exactly how all this stuff works in no time at all.

It is nice that the ultimate work book in this course was dedicated to writing such a useful and reusable tool and that the data generated by such a tool will certainly be used in GP3 where it will be used in conjunction with a more complete graphics engine. When using complex pixel shaders with multiple passes (as we will be in GP3), reducing overdraw has once again become a premier design goal. The more wonderful and complex these programmable pixel shaders are, the more strain is placed on the 3D hardware for each pixel that it renders. Using a PVS we can make sure that our 3D hardware does not grind to a halt spending most of its time performing multiple passes on pixels that are occluded and can not even be seen. If it has not become apparent to you yet, let me just say that by developing this tool and the means to render its data, we have overcome a huge obstacle in game engine development. The ability to render only the small region of the scene that is currently visible from the camera. Our games can now spend that GPU processing power rendering high detail objects in the immediate vicinity instead of rendering an abundance of low detail objects that can never been seen but are rendered anyway.

The lab projects in this course have certainly become rather large at times but we have many re-usable modules in place now and have acquired many techniques that were necessary to learn before we could even start to construct a real graphics engine. We are certainly ready for Graphics Programming with DirectX Part III.